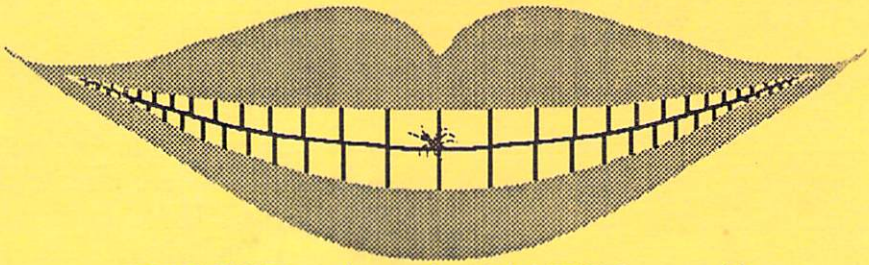
K.I.S.S.E.D 680x0

K Keith's
I Interactive
S Symbolic
S Screen-
E Editing
D Debugger

for all of the Commodore Amiga family of computers



KISSED simply *EATS* bugs!!

Designed and programmed by Keith Enge
(c) 1992 TTR Development, Inc.
6701 Seybold Rd.
Madison, WI 53719

This manual and all programs, artwork, sounds, animations, and utilities provided herein, with exception of the Amiga Workbench programs, are entirely copyright 1992 by TTR Development, Inc., 6701 Seybold Rd. Madison, WI 53719. All rights are reserved.

The programs and the documentation are sold "AS IS" and without warranties as to the performance, merchantability or fitness for a particular purpose. Sale of this software conveys a license for its use on (1) one computer owned and operated by the purchaser. Copying this software or documentation by any means whatsoever for any other purposes is strictly prohibited and is against the law.

No part of this manual may be copied, reproduced, translated or reduced to any electronic medium or machine readable form without the prior written consent of TTR Development, Inc.

The following copyright and licensing information refers only to the Amiga Workbench files contained in this package.

Amiga Workbench V1.3 and V2.0
Copyright 1985, 1986, 1988, 1989, 1990, 1991,1992 Commodore Amiga, Inc.
All rights reserved.

Your use of the diskette, software, and the manual indicates your acceptance of these terms and conditions:

1. Copyright: These programs and the related documentation are copyrighted. You may not use, copy, modify or transfer the programs or documentation except as expressly provided in this agreement.

2. License: You have the non-exclusive right to use any enclosed program on (1) one computer. You may physically transfer the program from one computer to another provided the program is used on only one computer at a time. You may *not* electronically transfer the program from one computer to another over a network or bulletin board system. You may not distribute copies of the program or documentation to others. You may not de-compile, dis-assemble, reverse engineer, modify or translate the program or documentation. All other rights and use not specifically granted in this license are reserved by TTR Development, Inc. and/or Commodore Amiga, Inc.

3. Backups/Transfer: You may make (1) one copies of the program solely for the purpose of making a backup of your original disks. You must reproduce and include all copyrights on the backup. If you transfer ownership of this program and documentation, the other party must agree to the terms and conditions of this agreement and must complete and return a registration card to TTR Development, Inc. At the time of the transfer, you must at the same time transfer all documentation and destroy all backup copies.

4. Terms: This license is effective until terminated. You may terminate this license by destroying the program, disks, and documentation and all copies thereof. This license will also terminate if you fail to comply with the terms and conditions set forth in this agreement. You agree upon such termination to destroy all copies of the program and its documentation.

TRADEMARKS:
Amiga is a registered trademark of Commodore Amiga, Inc.
AmigaDOS is a registered trademark of Commodore Amiga, Inc.
KISSED ©1992 by TTR Development, Inc.
Concept, Design, and Programming by Keith Enge.

Table of Contents

preview:

	table of contents	3
	screen editing	6
	special keys	8
	configuration program	11
	executable commands	13

commands:

syntax

. ,	Delimiters separating 1st and 2nd parameters	15
<	Delimiter separating 3rd from 1st parameter	16
/	Indicates 2nd parameter is length not end	17
%	Preceding number is decimal not hexadecimal	18
{ }	Math between { } considered as single number	19
rubout	Remainder of line past rubout is ignored	20

memory

space	Display memory as hexadecimal values	21
O	Display memory as ASCII	23
W	Change width (word size) of memory display	25
;	Change hex memory in current word size	26
:	Change hex memory but override word size	28
" '	Change ASCII memory	30
X	Terminate memory change (; or :) command	31
@	Create/remove "window into memory"	32
U	User variables or allocated memory blocks	34

registers

R	Display registers, pc, flags, opcode	36
a d p	Change registers, pc, flags	38
I	Display breakpoints and their counters	40

Table of Contents (continued)

b	Change breakpoints and their counters	42
Y	Regain last registers, etc, breakpoints, etc	43
[ ]	Pointer indirection into memory	44
	execution	
G	Go execute from pc	46
GG	Go only until logic fork	46
GH	Go hook a currently executing task	47
G]	Go until return address on stack	47
T	Trace once	48
\	Go execute until next sequential opcode	49
K	Skip current opcode to next sequential opcode	50
F1-F5, F10	Set boolean test on that breakpoint	51
J	jump in and set a breakpoint "on the fly"	54
N	Create/filter/access trace buffer	55
^	Lock register display in place	57
@	Create/remove "window into memory	32
_	Target task addr / segment display & Lists	89
	assembly/disassembly	
L	Disassemble memory	58
L>	Disassemble program to disc in source form	60
=	Assemble line to memory	62
'	Symbol delimiter	64
?	Symbol table search	66
*	Program counter address	67
	blocks	
H	Hexadecimal search	68
S	ASCII string search	69
V	Verify memory block	71

Table of Contents (continued)

M M+ M-	Memory block move intelligently/up/down	72
Z	Memory block fill	74
	miscellaneous	
>	Load program, read/write file	75
!	Load debugging file, do it all/single-step	78
&	Set/display target task stack size	80
%	Preceding number is decimal not hexadecimal	18
$ #	Convert dec/hex or hex/dec and display	82
(	Floating point decimal input	83
)	Floating point decimal output	84
~ + -	Do math and display results	85
|	Change scrolling rate	87
‘	Changes to KISSED's palette	88
_	Target task addr / segment display & Lists	89
P PM	Toggle printer / "print" to memory	90
Q	Quit KISSED	92
>>	Create and/or access clone	93
postview		
	warnings and tips	95
	quick reference list	104
	screen editing keys	106
	special keys	107
	index	108

Default Screen Editing Keys

KISSED uses its own code to do screen editing to avoid disturbing system resources. Since most users are familiar with the micro-EMACS text editor, KISSED's screen editing keys are the same as those used there with some differences in implementation. The major difference is that KISSED operates in typeover rather than insertion mode for ease in screen editing. If the user wishes, these keys can be reassigned using the included configuration program. The default definitions follow.

left arrow right arrow up arrow down arrow	Move the cursor and screen wrap in all 4 directions
tab	Nondestructive forward tab to next hex value
ctrl-A	Move cursor to column 1
ctrl-E	Move cursor to past the last non-space in line
backspace	Delete character before cursor, "rubberbanding" line
del	Delete character at cursor, "rubberbanding" the line
ctrl-O	Opens a space at cursor, "rubberbanding" the line
ctrl-K	Delete to end of line
ctrl-alt-K	Delete to start of line
ctrl-S	Delete to end of screen
ctrl-T	Toggle between overwrite and insert mode for that line until you cursor off that line
ctrl-C	Copy the current line to the line below
ctrl-P	Paste text on screen at cursor. To use it, press ctrl-P to indicate the place, then cursor to text to insert and ctrl-P over it (optionally, then cursor somewhere else and ctrl-P some more). Any key not a ctrl-P or a cursoring key (arrows, shifted left arrow, TAB, ctrl-A, ctrl-E) ends the paste process and places the cursor beyond the pasted text.
ctrl-D	Delete through any rubout character past the cursor. This is useful in getting to the unexecuted half of

	command lines that are split by the rubout character.
ctrl-alt-D	Delete until the next space
ctrl-F	Override the fixed format in register and breakpoint display lines (entering a 'symbol' does it also).
ctrl-L	Splits the cursor line into two lines at the cursor if and only if the cursor is not in the first or last four columns of the screen and the line is not split already. The resulting double line will behave like a single line while editing. For example, insertion or deletion of characters in the first line extends the "rubberbanding" to the second, line deletion deletes both, etc. The "stretch" hilited m's that pad the first line can be typed over to gain more usable length in that line. The "more"'s that end the first line and start the second, however, can NOT be typed over; they can only be removed by deleting the whole double line or deleting to the end of the line while the cursor is in the first line or the "mmmmmmore"'s.
shift down	Insert line, moving remaining lines down
shift right	Delete line, moving remaining lines up
shift up	Move cursor to screen top, clear screen
shift left	Move cursor to top, do NOT clear screen

ctrl-N When pressed anywhere within a disassembled line, that line is prepared for easy editing by eliminating the extraneous stuff and placing an = after the address.

ctrl-R Resume address is placed on an inserted line for reuse. For example, suppose a display of memory (space command) was either ended normally or a scrolling display aborted, ctrl-R displays the address needed to restart from where the display ended. It works equally well with the O ; : ' " L = H S V commands as well as the result of math operations (see also alt-R which follows).

alt-R Places the "resume address" (see ctrl-R above) at the cursor, overwriting whatever was there.

ESCape allows the normally non-printable characters (all 256) to be typed on the screen. It can be used in two different ways.
1) If the ESCape is followed by three decimal digits, the ASCII equivalent of that three digit decimal number will be placed. For example,
065 = A while 193 = 065+128 = hilited A.
2) If the ESCape is followed by a period and then two hexadecimal digits, the equivalent will be placed. Therefore .41 = A and .C1 = hilited A

help Merely displays, as non-destructive "popup", a list of the sixty some valid commands and the memory available. For the 68030, the supervisor registers are also shown. If the keypress that ends the "popup" is a "command", the help text for that command from S:KISSED.HLP will now appear as another "popup".

ctrl-X Exchanges screens. KISSED's screen is sent to the back until another key is pressed (normally the target screen will then appear). This saves cycling with the "depth" gadget.

ctrl-B Temporarily blanks the screen until another key is

pressed (merely a screen-saver function).

ctrl-V — Displays an ASCII table, the escape key sequences to produce non-ASCII characters, and the processor flag combinations for various condition codes. This display is a non-destructive "popup".

ctrl-W — What the various values in the cursor line are is shown. PC relative addresssing gives the address (or symbol) pointed at. 'label' gives the longword for that label. Hexadecimal numbers preceded by a $ are displayed as their ASCII equivalents. If it is used in a breakpoint display line, it displays their symbolic values.

alt-2 — Displays the snapshot of the "window into memory" established by the @ command. Changes are hilited.

ctrl-2 — Displays the previous values of the snapshot of the "window into memory" established by the @ command.

ctrl-alt-2 — Clears (undoes) the "window into memory" established by the @ command.

ctrl-\ — If the cursor is in a disassembled line, pressing this combination will set a "point and shoot" breakpoint there and starts execution. If the target program is currently executing, a "jump in on the fly" breakpoint is placed (see J command).

alt-3 — Toggles the assembler/disassembler between the 68030 instruction set, the 68000 set, and the 68000 set with the old style addressing mode formats; ie $xxxx(A4) rather than the new ($xxxx,A4) . Of course, this key doesn't really change the processor, but it does affect the disassembler and assembler commands (L and =) so that a 68000 based system can assemble and disassemble 68030 instructions. Conversely, a 68030 system can choose one of the 68000 modes to restrict the assembler and

disassembler to only those instructions valid on the 68000.

return or enter — Execute the line that the cursor is in as a command line. Nothing really happens until this key is pressed when the cursor is somewhere in a line; then that whole line (before and after cursor) is executed as a command line. Thus, any changes made while cursoring around the screen do nothing except change the screen until return (or enter) is pressed; then that cursor line is executed.

ctrl-return — Does a carriage return, linefeed just like the RETURN key but doesn't cause the line to be executed. It is thus equivalent to a down arrow and then a go to line start (ctrl-A) except for one special case. That special case is the reason for this command and occurs when the cursor is in the last line of the screen and you want to scroll the screen without executing that last line. Without this keypress, you would have to create a blank line either by deleting the line or cursoring up and then deleting a different line. This command allows the screen to be preserved and merely scrolls.

Read and understand the above RETURN key note; it accounts for much of the power and ease of use of the debugger. Let KISSED do the typing for you and just edit the results.

The left mouse button can be used to reposition the cursor.

Reconfiguring the editing keys

The CONFIGURE program allows the user to redefine the 40 "editing" keypresses (those keys don't produce a typed character) from a list of 120 possible keypresses. On the left side of the screen will be a display of the 40 actions and their current keypresses; on the right are the 120 available keypresses (the already used ones are hilited). To change a key, use the ARROW keys to move the cursor to the action's current key and press RETURN. The corresponding key on the right side will have its hiliting removed and the cursor placed there. Use the ARROW keys to move to the replacement key and again press RETURN. This key will be hilited and the left side of the screen will be updated with the new key and the cursor moved back there. You are now ready to change another key. While choosing replacement keys, you aren't allowed to choose a key already in use. If you want to reuse a key, you will have to choose the replacement for that key first to free it up. The arrow keys wrap vertically and horizontally. When you are done, pressing HELP will save the new setup as the file KISSED.KEY. Pressing DEL will revert to the old setup. If the program is later run again, the new setup will be read from the KISSED.KEY file, modified, and then resaved. If the user wants to revert to the original default keypresses in either CONFIGURE or KISSED itself, just delete or rename the KISSED.KEY file first. For KISSED to find and use the KISSED.KEY file, it must be in the S: directory. The CONFIGURE program, when first run, writes KISSED.KEY to that S: directory; rerunning the CONFIGURE program modifies that file in S:.

Note the alt-ESCape keypress, although listed as an available keypress, should not actually be used as a replacement for some other keypress. It was included as an available keypress because it is a keypress recognized by the keyboard. However, that keypress has a special significance to the console.device as a CSI (control sequence introducer). Thus, it will "eat" at least the

subsequent keypress (and potentially some additional ones) in attempting to build a valid control sequence. Therefore, don't choose alt-ESCape as a replacement editing keypress or use it from within KISSED; it won't "break" anything but it probably won't be useful either.

Executable commands

A command line is defined as the line in which the cursor was located when the RETURN (or ENTER) key was pressed. This line could have been anything; newly typed in, a previous command line (possibly re-edited), or a line previous displayed by KISSED (possibly subsequently edited); KISSED makes no distinctions.
Thus, for example, a display of memory, registers or breakpoints can have the displayed values changed by editing and then pressing RETURN makes it a command line and the changes are actually made.

The command line uses a post-processor; a command works on the value immediately preceding it in the line (with no intervening spaces) and allows multiple commands per line. These multiple commands can be typed in free form as a continuous list and don't need to be separated by spaces (or anything else). Commands concerned with multiple parameters must be in a continuous string with no separating spaces since space following a number is the display memory command (for example, the multiple parameter move memory command M, which moves the block between xxxx and yyyy to the location zzzz, has the format zzzz<xxxx.yyyyM .
If an invalid command, or a command without required parameters, is found, KISSED aborts execution of the line at that point (any previous valid commands will already have been executed) and echoes the command line with the error hilited.

All numeric values are hexadecimal.

A description of the valid commands follows. They are roughly grouped into seven categories; syntax, memory, register,execution, assembly-disassembly, block, and miscellaneous. The list of valid commands, a few of which have multiple purposes is G thru Z, space ! @ # $ % ^ & * ( ) _ - + = ~ ` [ ] { } ; : ' " | < > , . / ? a b d p

KISSED is a postprocessor; when it expects a hexadecimal number, it merely keeps reading digits until it reaches a non-digit. That non-digit character then becomes the command which operates on that preceding number.

Although KISSED is case sensitive, the user may use upper and lower case interchangeably; KISSED merely converts everything except symbols and ASCII strings to upper case. Symbols are enclosed by single quotes (') , while ASCII strings are enclosed by either single or double quotes. This is particularly apparent when KISSED perceives an error in a line and echoes the line with the error hilited; the echoed line will be in upper case, except for anything within quotes.

A scrolling display can be paused by the space bar, then restarted by pressing it again. RETURN or any other non-printing keypress will scroll the display a "page" at a time; any other keypress during a scrolling display aborts the display. Holding down the right mouse button also pauses the display.

A . (period) separates the 1st and 2nd parameters in a command. These are often the start and end, respectively, of a range of memory. If the / command is used following the 2nd parameter, they become the start and length instead. If a math command (+, -, or one of the ~ operators) follows the 2nd parameter, then the period separates the 1st and 2nd arguments.

A , (comma) can be used interchangably with a period; KISSED makes no distinction between them.

For two commands, disassemble and display ASCII strings (L and O, respectively), the 2nd parameter is a line count rather than an end. This is an inconsistancy in the user interface but a line count is much more useful in these two commands than an end since variable opcode or string lengths make the end usually unknown or unimportant.

examples:

1234.5678	Displays the memory range from 00001234 thru 00005678
1234.AF+	Adds 00001234 to 000000AF and displays the result
23456.5L	Disassemble 5 lines at 00023456.

The < separates the 3rd parameter of a command from the 1st one in commands that need three parameters. Note that this is, in a sense, a misnomer since the 3rd parameter will actually be the first parameter listed (ie 3rd<1st.2nd).

This third parameter is usually the start of a 2nd memory range; for example, if two blocks of memory are to be compared with the Verify command, the 1st and 2nd parameters give the start and end (or length/) of one block while the 3rd parameter gives the start of the other block. Occasionally, the 3rd parameter will be used for something else in commands which have other needs for three parameters.

If a 3rd parameter is given for commands which only need two, it will not produce an error, it will merely be ignored.

examples:

30000<1234.2AB9M	Moves (copies) the memory block which starts at 00001234 and ends at 00002AB9 to a new location 00030000.
4<10000.C/	Displays 4 blocks of memory in C (twelve) byte sized chunks, starting at 00010000.

The / indicates that the 2nd parameter that immediately precedes it is the length of the memory block defined by the 1st and 2nd parameters and not its end.

examples:

30000.1400/	This is equivalent to 30000.13FF and would display 1400 (5120 decimal) bytes of memory.
20000<30000.1400/V	This would verify that those 1400 bytes are identical to those at 00020000 and would display the differences.

The % command signifies that the preceding parameter is to be interpreted as a decimal rather than a hexadecimal value.

examples:

10000.100%/	This would display one hundred bytes of memory starting at 00010000. Without the %, 256 bytes would be displayed.
10000%.100%/	This would also display one hundred bytes, but this time it would start at 00002710 (10000 decimal).
10000%.100/	256 bytes from 00002710 this time.

Enclosing a math operation in { } causes the result of the math operation to be considered as a single parameter. Only the following math operators are allowed; +, -, ~+, ~-, ~*, ~/, ~+F, ~-F, ~*F, ~/F, ~&, ~|, ~^, ~< and ~>. Note that signed multiplication and division are not allowed; neither, of course, are not (~~) and binary display (~%).

{ } can be nested indefinitely.

examples:

{1234.70000-}.100/	256 bytes (100 hexadecimal) of memory will be displayed, starting at 0006EDCC which is 00001234 subtracted from 00070000.
{4660%.70000-}.100/	Same as above since 4660 decimal is 1234 hexadecimal.
{10000.{4.346~*}+}.{8.6A8~/}/	This too is a memory display whose length is D5 which is 6A8 divided by 8. The start of the display is 00010D18 which is the sum of 10000 and the product of 4 and 346 .

KISSED uses the last syntax command, a rubout, to separate non-executable portions of a displayed line from the portions which could be executed if the RETURN key was pressed while the cursor was in the line. If the user wishes to make the line a command line, the rubout can merely be removed.

If the user wants to place a rubout in some line at the cursor, an escape sequence is used; press the ESCape key, period, 7, and finally F (alternately, the sequence ESCape, 1, 2, 7 could also be used).

The rubout character is a command line terminator; everything following it in the command line will be ignored. It is used to start the output line of the Verify command, the disassembly List command, and various messages. It is used there to protect the user from potentially nasty things happening in case the RETURN key is accidentally pressed while the cursor is in one of these output lines. If, however, the user wants to use these lines as command lines, the rubout can be edited out. The rubout also is used in the output of the memory display space command (no pun intended) to separate the displayed memory from its ASCII representations and, by inserting the rubout, KISSED thus allows these output lines to be used as command lines with no modifications.

Sometimes, the user may want to delete everything in a line up to and including the rubout, the ctrl-D keypress does this. One example of its use is when editing a displayed memory line. Rather than editiing the hexadecimal values, the user can place the cursor after the initial address and press ctrl-D. If a quote is now inserted and also replaces the final rubout, the result is a change ASCII command and the sixteen characters withing the quotes can be edited.

Memory space

The space command (no pun intended) is used to display a block of memory delimited by the 1st and 2nd parameters. If there is a 3rd parameter, it is the repeat count; it is useful mainly in separating and displaying data structures.

The memory block(s) will be displayed as follows. Each line will have its starting address, a semicolon, and then 16 bytes in the current word width (as set by the W command) separated by spaces. Following those 16 bytes is a rubout and then the ASCII equivalents of those 16 bytes.

If the current word width is word or longword, the starting address will be forced to even by truncation. If this occurs, the addresses will be shown hilited in red.

In the ASCII equivalent portion of the display line (or, for that matter, anywhere in KISSED), the displayable ASCII values ($20 - $7E) are displayed normally. $7F becomes a rubout character while $00 thru $0F become a backlit 0-9 or A-F. $10 thru $1F become underlined versions of those backlit 0-9 or A-F. For values $80 or above, the equivalent $00-$7F value is shown hilited in red.

Note that the memory is displayed in a form that is itself a valid command line (see the ; command) which allows easy editing of displayed memory.

Note also when the display is ended, either normally or a scrolling display was aborted with a non-space bar key, the ctrl-R or alt-R keypress will produce the resumption address.

examples:

10000.10022	Display the sixteen bytes in 00010000 thru 0001000F on one line, then a second sixteen 00010010 thru 0001001F on a second line, and finally the remaining three bytes 00010020 thru 00010022 on a third line.
10000.23/	Identical with example above.
3<10000.23/	The separate blocks displayed, 00010000 thru 00010022, then 00010023 thru 00010045, then 00010046 thru 00010068

The O command outputs (displays) memory as ASCII strings. The 1st parameter is the starting address and the 2nd parameter is the number of strings desired. If there is no 2nd parameter, it defaults to one. If there are three parameters, the format is addr<start.end where addr is the starting address, start is the number of the first string after addr to display, and end is the last string display. Note that thus start<=end; if start=end, then only that string will be displayed.

Each string will be a maximum of 70 characters long, enclosed in single or double quotes (' or ") and preceded by its starting address. If there are ' or " in the string, the opposite type will be used to enclose the string. If both types would be within the string, the string will end before the first occurance of the second type.

Other things can cause the ending of a string besides the presence within the string of the enclosing delimiter; these are the typical string terminators. They include some combination of carriage returns and line feeds, a null, or a character with its high bit set. These string terminators will be included in the string; if there are multiple consecutive terminators of the same type, they will all be included in the string, a new string will start with the next normal character or terminator of a different type. There is one exception to this is that any nulls that follow a string terminated by non-nulls will be added to that string anyway.

Note that the string is displayed in a form that is itself a valid command line (see ' and " commands) which allows easy editing of displayed memory.

Note also that the ctrl-R and alt-R keys will produce the address beyond the last string displayed for resumption of the display.

examples:

12345O	Displays one string at 00012345.
12345.4O	Displays 4 strings starting at 00012345.
12345.O	Displays strings starting at 00012345 forever. This scrolling will continue until another key is pressed (the usual pause, page, or abort keypresses).
12345<4.F	Displays the 4th through 15th strings.
12345<6.6	Displays only the 6th string.

Note that the above command 12345.O is somewhat strange in that the period says there is a 2nd parameter when there isn't one since the O command follows it immediately.

The W command changes the memory width. 4W means four bytes per value (longword), 2W means two bytes (word), and 1W is a byte. This value affects both display of memory and changing memory. If the user displays memory using the space command (see above), the display width is the last width specified. The default width upon entry to KISSED is 2W. Similarly, when using the change memory command (see ; below), it expects values of the chosen width. Longer values will be truncated (leftmost bytes ignored) and shorter values will be right justified (zero padded from left).

Care in memory width should especially be taken when reading and/or writing to the dedicated hardware addresses at $00BFxxxx and $00DFFxxx. Many of these are byte addresses and should always be accessed as such with a width of 1. Strange and wondrous things can happen to the operating system if one is accidentally accesses a byte above or below that intended. In addition, most are read only or write only. They also really should be avoided but if one really insists on poking around there, the : command should be useful.

Beware of not knowing the current memory width when changing memory. Values not of the expected width will be padded with zeroes or truncated as needed. Modifying displayed memory largely eliminates this problem since the visual reinforcement is right there on the screen (including address hiliting if address truncation has occurred). Note the displays that result from the Search, Hex search, and Verify command output in bytes, regardless of the memory width. Therefore, if one of its output lines is to be modified for use as a command line by removing the "rubout", the single colon in the line temporarily forces the width to 1 (bytes).

The semicolon specifies the enter memory command. It takes only one parameter, the value before the ; is the starting address. All values following the ; (separated by one or more spaces) are the values to enter. The values entered will be in the current memory width (see W command above) and will be either truncated on the left if too
long or zero padded from left if too short.

Note that if one wants to enter more values than will fit on a command line, a ctrl-R or alt-R will produce the address past the last entry for resumption on another command line.

Note also that to make KISSED more "bullet-proof", you won't be allowed to enter memory in locations within KISSED.

examples: (note that in all the examples that follow, the W command command that starts the line isn't needed, it is merely there to show what the current width is)

1W 12345; 34 56 789AB	Since the word width is bytes, the values entered at 00012345 will be 34, 56, and AB.
2W 12345; 34 56 789AB	Since the word width is words, the vaues entered at 00012344 (12345 forced even) will be 0034, 0056, and 89AB.
4W 12345; 34 56 789AB	Since the word width is longwords, the address will be talues entered will be 00000034, 00000056, 000789AB.

1W 12345; 34 56 14568; 58 9A Since the word width is bytes, 34 and 56 will be placed, then, just before the 68 would be placed, the following ; is found and the 14568 instead becomes a new changed memory command and the values 58 and 9A are placed at 00014568.

Note that the above is an example of multiple commands per line.

Care in memory width should especially be taken when reading and/or writing to the dedicated hardware addresses at $00BFxxxx and $00DFFxxx. Many of these are byte addresses and should always be accessed as such with a width of 1. Strange and wondrous things can happen to the operating system if one is accidentally accesses a byte above or below that intended. In addition, most are read only or write only. They also really should be avoided but if one really insists on poking around there, the : command should be useful.

Memory :

The : command is a change memory command like the ; command above, except that the memory width is temporarily ignored. The temporary width used is determined by the number of colons (1=byte, 2=word, 4=longword). Note that KISSED uses this in the displays that result from the S, H, and V commands.

Care in memory width should especially be taken when reading and/or writing to the dedicated hardware addresses at $00BFxxxx and $00DFFxxx. Many of these are byte addresses and should always be accessed as such with a width of 1. Strange and wondrous things can happen to the operating system if one is accidentally accesses a byte above or below that intended. In addition, most are read only or write only. They also really should be avoided but if one really insists on poking around there, this : command should be useful.

Note that if one wants to enter more values than will fit on a command line, a ctrl-R or alt-R will produce the address past the last entry for resumption on another command line.

Note also that to make KISSED more "bullet-proof", you won't be allowed to enter memory in locations within KISSED.

examples: (note that in all the examples that follow, the W command that starts the line isn't needed, it is merely there to show what the current width is).

2W 12345: 89 ABCD 23454; 34 3456 Although the memory width is words, the single : temporarily forces the bytes 89 and CD to be entered. The ; in the next command means that the width is used so 0034 and 3456 are entered at the new address.

12345:: 89 ABCDE 45678:::: 89 ABCDE Regardless of the memory width, first the words 0089 and BCDE are placed at 0012344 (truncated), then 00000089 and 000ABCDE are placed at 00045678.

The " and ' commands enter an ASCII string into memory at the address given by the parameter. The delimiters that enclose the string are the single or double quotes. If there is a quote in the string, the other type must be used as the delimiter. Therefore, strings can not contain both types of quotes simultaneously. Note that this is a double use of the ' character, it is also used in symbols (see ' command). Non-printable characters can be entered by using the escape key and three decimal digits or the escape key, a period, and two hexadecimal digits.

Note that if one wants to enter more values than will fit on a command line, a ctrl-R or alt-R will produce the address past the last entry for resumption on another command line.

Note also that to make KISSED more "bullet-proof", you won't be allowed to enter strings in locations within KISSED.

examples:
12345"Hello"
12345'say "Hello" for me'
12345"He's the one who said it"

The X command is the terminate memory entry command. There is actually only one time this command is needed. It is when there are multiple commands on a line and a change memory command (either ; or :) is immediately followed by a display one memory value command (space command with only one parameter). An example is shown below.

example:

2W 1234; 45 67 9ABC;123 4567 X 3456

> First the words 0045 and 0067 are placed at 00001234. The byte BC isn't placed it is followed by a ; so instead 0123 and 4567 are placed at 00009ABC. Finally the word at 00003456 is displayed (without the X, the word 3456 would be placed at 00009AC4 instead). On the other hand, if the 3456 had been 3456.20/ instead, the X would not have been needed.

2W1234;45 67 9ABC;123 4567 X3456

> KISSED normally doesn't need spaces between commands; the above line is exactly the same as the previous example. Spaces are only needed to separate the values in the memory entry or for the space command to get a memory display.

The @ is the "look At window into memory" command. It is used to create the window and also to toggle its enabling/disabling. The purpose of this "window into memory" is as follows. When execution of the target program returns to KISSED after a breakpoint or trace, etc, the display of memory values in the window will be automatically updated to reflect any changes since the last return to KISSED. Changes in values will be immediately obvious because they are hilited. The display of the window is a non-destructive popup at the top of the screen which will disappear when any key except two, ctrl-2 and alt-2 is pressed. The ctrl-2 keypress can be used to see the previous values; the alt-2 keypress reverts to the new ones. When a key is finally pressed which is not one of these two, the new values are saved so they will becomes the new previous values for when the window is next displayed.

Note that the two keypresses, alt-2 and ctrl-2, can be pressed at any time to call up the window, not just when a return to KISSED calls up the window automatically. Normally the window is looking at variables in a stopped (non-executing) target program so it will be unchanged. However, if the target program is executing or the variables being looked at are operating system variables, the values can be constantly changing and each time the window is called up a new "snapshot" of them is taken.

The window is removed (not merely disabled) by a ctrl-alt-2 keypress.

The @ command either has three parameters or one. If there are three, the format is width<start.end@ where start and end define the memory range of interest and width is the display width (1=bytes, 2=words, 4=longwords). Each line of the window can display up to 16 bytes of the range. Each new three parameter @ command adds one or more lines to the window (the maximum size of the window is 10 lines; a message will be displayed when

all lines are full). The ctrl-alt-2 keypress mentioned above completely empties the window so the the user can start again. If start is 0 through 7, instead of pointing at memory address 0-7, that address register is used as a pointer and that part of the display is "locked" to it (if the value of that register changes, the display repoints also).

If there is only one parameter, the possible commands are 0@ and 1@ which disable and enable the window, respectively. Disabling is done when the user doesn't want the window popping up every time KISSED is returned to, but, on the other hand, doesn't want to clear the window (with ctrl-alt-2) because it will be useful later. Note that while the window is disabled by 0@, the alt-2 and cntl-2 keypresses can still be used to take manual "snapshots".

The U command defines and uses user variables and memory blocks; there are ten of them, designated U0, U1, U2, ... , U9. The commands take either one or two parameters. If there is one parameter, the format is valueUx to set the value of a user variable x (initially they are all set to zero). Subsequently, the variable Ux can be used anywhere a normal hexadecimal number is used (parameters, breakpoints, etc.). Using the bare Ux followed by a space as a command just displays its current value. Attempting to set a given variable Ux when that same Ux is currently in use as a memory block (see below) will "error out".

If there are two parameters, it is a memory block request and takes the format size.typeUx where size is the block size in bytes and type is either 2, 1, or 0. Each type results in a zeroed "public" memory block; 2=fast, 1=chip, 0=either fast or chip (fast is tried first). The command will "error out" if the allocation couldn't be made. It will also error out if Ux is currently being used as a different memory block; however, it it is merely being used as a variable, it will supercede its use. If Ux is now used as a normal hexadecimal number, its value will be the start of the memory block; a bare Ux followed by a space as a command will display the start and length of the block.

Note there is a special case of the one parameter command which is normally valueUx . If the Ux is currently being used as a memory block, the command will "error out" as mentioned above. If, however, the value is zero or one (0Ux or 1Ux), these have special meanings. 0Ux results in the deallocation (freeing up) of the memory block and Ux reverts to zero so the user can't use it to point to the now vanished memory block. Attempting to use the deallocated memory block anyway, via some other method, will inevitably eventually crash the system. 1Ux results in the permanent allocation of that block; Ux now becomes merely a variable pointing at that block. Even when quitting KISSED, the

block will remain. This is mainly useful in preserving patches in target programs left executing when quitting KISSED, otherwise a crash again would result.

Loading a non-executable file without specifying a read address (the 0>filename command) results in the use of one of these U0-U9 memory blocks and will error out if all are in use. Loading an executable file (>filename command) also uses one but only temporarily and then immediately frees it up again (however, one did have to be available at that time). Similarly, loading an autoexec file (! command) also temporarily uses a block until the file is done executing.

Any allocated memory blocks will be automatically deallocated when the user quits out of KISSED (Q command); the user doesn't have to do it manually (exception, see 1Ux discussion above).

If memory blocks are used to create a target by hand assembly, remember to allocated space for a stack and point A7 at its top.

Registers R

The R command is the display target registers command. This command displays 4 lines. The first line consists of a d: followed by the eight data registers. The second line consists of an a: followed by the eight address registers. The third consists of a pc: followed by the target program counter, then a usp: followed by the target user stack pointer, an ssp: followed by the target supervisor stack pointer, and finally a t_s__iii.___xnvzc: followed by the state of the target status registers flag bits. The fourth line is a disassembly of the opcode at the target PC. For the 68030 with its extra stack, the current stack pointer is in A7 of the a: line while the two other stacks are shown on the pc: line; in addition, the flag list is ttsm_iii.___xnvzc.

There is other important additional information available in this display. The values of D0-D7, A0-A7, PC, USP, SSP, and the individual SR flags are hilited if their values have changed since the previous breakpoint, trace, etc was executed. This gives instant visual reinforcement of any changes that have occurred and can be a definite help in solving the age-old "who clobbered D3 ?" question. In addition, besides the 0 and 1 bits of the target SR being hilited if they have changed, if the bits are currently 1, the corresponding letter in the t_s__iii.___xnvzc is hilited to reflect this.

If the disassembled opcode is at the current program counter, the disassembled fourth line contains additional information. If the opcode is a Bcc, DBcc, or Scc (or TRAPcc for 68030), a YES or NO will appear towards the end of the line depending on the state of the processor flags and which condition code is specified in the opcode. For a Bcc, a YES means the branch will be taken; for a DBcc (until loop), a YES means the loop will abort (it says nothing about the D register counter); for Scc, a YES means the byte will be set to $FF. If the instruction uses addressing modes with displacements or indexes, those effective addresses will be

evaluated using the current register values and the result displayed. Note that the new 68030 memory indirect addressing modes which fetch pointers from memory and then do yet a further fetch, must do the fetch now to evaluate the effective address. Therefore, if the registers used are garbage, the fetched pointer will be garbage and problems can result now when the opcode is disassembled, just like they would occur when it is executed. KISSED tries to protect itself and the user but can only do so much. Finally, any JSR d16(a6) or JMP d16(a6) to functions in the exec, intuition, graphics, or dos libraries will display the name of that function.

The a, d, and p commands aren't really entered by the user; they are merely the first letter on the three register lines displayed by the R command (see above). The values of the registers displayed in a line are changed by cursoring up to the value and changing it and any others desired in that line. Then, when the RETURN key is pressed, the changes are made. As usual, nothing happens until the RETURN key; if the values are edited but the line is cursored off without a RETURN keypress, the values are NOT changed unless the user later cursors back to the line and presses RETURN. Note that the changes normally only affect one line; if changes are made in another line, they must have their own RETURN keypress to be changed. However, changes to A7 will affect the usp or ssp in the next line; conversely, changes to the supervisor bit and/or usp/ssp (whichever is applicable) will affect A7. Changes to the flag bits of the target SR are made to the 0 and 1 bits themselves, not to the letters which are hilited to echo their state. The high nibble of the SR (t_s_ or, for the 68030, ttsm) will always be 0000 and probably should never be changed. In fact, KISSED won't let the user change the tt bits. Setting the s bit will change to supervisor mode and any exception in supervisor mode on the Amiga, including breakpoints and traces, will invoke the GURU. Setting the m bit will use the unitialized msp stack with disastrous results.

When editing these register display lines, the user may notice that the backspace, ctrl-I, del, ctrl-K, and ctrl-S keys are disabled thus limiting the user's changes to typeovers. The reason for this is that these lines are so packed with information that their format must be made rigid rather than the usual free format of other command lines. To successfully parse the line, KISSED expects a certain number of hexadecimal values in a specific order in these lines; therefore, the user's changes are restricted to replacing hexadecimal digits with other hexadecimal digits. No changes can be made to an executing target.

If the user wishes, this fixed format can be overridden in the current line if a ctrl-F is pressed while in the line. This is mainly useful in allowing the entry of symbols or embedded math rather than hexadecimal values (actually, typing the ' that starts a 'symbol' automatically does the override without needing the ctrl-F).

Care should be taken if this overriding is done to avoid changing the line so much that KISSED can't recognize it. The d: and a: lines must start with the a: or d: and have 8 values on the line. The pc: line must start with the pc: and then a space; the remaining values on the line will be found by searching for colons, then tossing any spaces, and finally using the numbers that follow.

68030 users note that the current stack will be in A7 in the a: line, the other two stacks will be in the pc: line.

The I command displays the five possible breakpoint addresses and their associated counters (word). The format is counter, period, address where counter is the number of times that the address of that breakpoint must be passed through before that breakpoint is activated. A 1 means that the breakpoint happens the first time that address is reached; a 2 means that the break occurs the second time; an FFFF means that it occurs the 65535th time. As a convenience to the user, when a counter has been decremented to zero, it is immediately reinitialized to 1. If either the address or the counter is zero, the breakpoint is not set. Thus, making its counter zero, allows a breakpoint to retain its value without being set. Values are hilited if they have changed since their previous use.

Breakpoints can even be set in the system ROMs; since this is physically impossible, their use is simulated by continuously tracing everything. This continuous tracing by setting breakpoints in ROM will drastically slow up everything and thus should be done only in two situations; while temporarily investigating the system ROMs or when this slowup is desired for slow motion execution. Note that this slowup will occur even if the ROMs are never entered by the execution; just the presence of the breakpoints will cause tracing to occur everywhere in the code, not merely if the ROMs are entered.

The periods between the counter and breakpoint address is hilited if there is a boolean test associated with that breakpoint. Similarly, the b itself is hilited if a boolean test will be made after each and every instruction, not just at a breakpoint. If this test is used, continuous tracing is necessitated which will drastically slowup the execution (by a couple orders of magnitude).

Pressing the ctrl-W while in a b: line will display any symbols associated with the breakpoints, one line per breakpoint. If there are multiple symbols for a given breakpoint, they will be listed on

indented separate lines.

Do NOT set breakpoints in code that will be executed in supervisor mode, the system will GURU since any exception in supervisor mode is a no-no.

The b command isn't really entered by the user; it is merely the first letter in the line containing the breakpoint display that results from the I command (see above). Breakpoints and their associated counters are therefore changed by displaying them, cursoring up and making changes, and then pressing RETURN to execute that line. Like the change registers commands (see a, d, p above), the line format is rigid; editing capabilities are restricted to typeovers. If the user wishes, this fixed format can be overridden in the current line if a ctrl-F is pressed while in the line. This is mainly useful in allowing the entry of symbols or embedded math rather than hexadecimal values (actually, typing the ' that starts a 'symbol' automatically does the override without needing the ctrl-F). No changes can be made to an executing target.

If the counters or the breakpoints are set to zero, that breakpoint isn't used. Therefore, breakpoints can retain their values but still be turned off by setting their counters to zero.

Pressing the ctrl-W while in a b: line will display any symbols associated with the breakpoints, one line per breakpoint. If there are multiple symbols for a given breakpoint, they will be listed on indented separate lines.

Do NOT set breakpoints in code that will be executed in supervisor mode, the system will GURU since any exception in supervisor mode is a no-no.

Breakpoints can't be set while the target program is executing except via the J command and ctrl- keypress, both which see.

If breakpoints are left in a target program still executing when quitting KISSED, the breakpoints will be removed.

The Y command exchanges the current target registers and breakpoints (and their associated counters) with their previous values. This will mainly be used when an execution didn't produce an expected result. By using the Y command to get the values from before that execution, the execution is, if effect, undone (at least as far as the register values are concerned) and so can be run again after any modifications with, hopefully, better results. Its most common use when tracing and the user traces one opcode too many and takes a branch to a subroutine. If the user doesn't want to trace through the subroutine, the Y command will "back out" of the subroutine and then a \ command (which see) can be used to regain control after the subroutine executes.

No changes can be made to the registers of an executing target.

The] command uses pointers for memory indirection. If the] is preceded by a number between 0 and 7, value currently in that address register (A0 to A7) is used instead. If no number precedes the], A7 (the stack) is used by default. If the number is 8 or greater, then the pointer is not obtained from an address register but is instead obtained from that address.

The [command is the same as the] command but uses the data registers (D0 to D7) instead of the address registers used in the] command. Again, like the] command, preceding numbers 8 or greater point into memory.

More than one] or [can be used in succession to get more layers of indirection. Thus, 4]] first gets the longword from A4, then uses it to point into memory to obtain the longword stored there. This second longword is requested value of 4]].

examples:

4].7] 4.]	Both of these have the same result, the memory between the addresses pointed at by address registers A4 and A7, respectively, is displayed.
3[[.800%+	The value displayed will be the sum of 800 decimal and the longword stored at the address in the D3 register.
12400]]]	This will display, in the current memory width, the value pointed at by the longword stored at the address pointed at by the longword stored at 00012400

There is a fairly common use of double indirection. It occurs when the user stops execution of a target program within a subroutine. Now, if rather than trace through the whole subroutine the user merely wants to regain control at the location to which the

subroutine returns when completed. Since the return address is on the top of the stack, using a ctrl-F on the line displayed by the breakpoint display command I, a breakpoint can be placed at]] and RETURN pressed to place it. Similarly, if the subroutine has already pushed some registers on the stack (or equivalently did a link opcode or both), the offset to the return address can be used like this {].20+}] . See also the G] command.

Note that indirection always fetches longwords; if the final fetch is to be a word or byte, use the shift operator to adjust that final fetch. For example, {23456].10~>}.a~* multiplies the word at 23456 by 10 (a longword is fetched but is shifted right 16 to become a word).

The G command goes to execute the target program. This command takes no parameters; execution starts at the target PC which can only be changed with the p command. This is intended to be a safety precaution in that it forces the user to see the other registers and breakpoints have the desired values while changing the PC. While the target is executing and before KISSED is returned to (via a breakpoint), this state is indicated by the cursor's "lips" being closed; in addition, several commands aren't allowed while in this state, although most others are available since the target and KISSED are multitasking.

The GG command is similar to the G command but an additional breakpoint (besides the user's normal five) is set at the next logic fork. KISSED searches forward sequentually from the program counter to the first instruction that will disturb the linear execution of code and places this breakpoint there. These logic forks are BRA, BSR, Bcc, DBcc, JMP, JSR, RTS, RTE, RTR, TRAP, TRAPV, RESET, STOP, lineA, lineF (for the 68030, the RTD and TRAPcc are added). Note that if the program counter is currently at a RTS, RTE, RTR, RTD, JMP, or a Bcc which will be taken (it will say "yes"), the GG command probably won't be appropriate since these instructions will change the program counter (but unlike JSR, BSR, DBcc, etc won't return back again) and thus setting the breakpoint foreward in memory won't be particularly useful, a trace or perhaps a normal G command would be better.

As a safety device, using the G (not GG) command without any breakpoints set (all have either their addresses or counters zeroed) will result in a "popup" warning which will be ignored only if a ctrl-G is pressed. Normally, while debugging, one would want some breakpoints set somewhere so this warning is provided. If, however, no breakpoints are wanted and KISSED is to regain control only by use of the "jump in on the fly" breakpoint (J command), this warning can be "disabled" by setting some dummy

breakpoint in some code that won't be executed again (such as the initialization code at the program's start).

The GH command is the "go hook" command which gains control of already executing tasks and takes the format, addrGH, where addr is the address of a task structure (which can be determined by the __ command, which see). When the task so hooked eventually becomes active again from the TaskReady or TaskWaiting lists, KISSED will gain control instead; until then, the target is considered to be executing (the "lips" are closed). When KISSED gets control that first time, the PC will be at an RTS; if the user traces once using the T command, the actual resumption PC will be reached (that first RTS is an artificial construction used by the hooking procedure so that that first breakpoint can be set even in ROMs (subsequently, breakpoints behave normally, see the I command). Don't make KISSED "eat its tail" by hooking tasks that it needs.

G] is equivalent to setting a temporary breakpoint at]] (or the equivalent 7]]), the address on the top of the stack. It is useful if one enters a subroutine inadvertantly and wants to set a breakpoint at where it returns.

This command causes a single trace of the instruction at the target PC. Although breakpoints may be set, they can still be traced through and their counters decremented. If a nonzero value precedes the T, continuous tracing is initiated; tracing will occur each time the spacebar is pressed (just as if the user types a T and a RETURN). Continuous tracing is stopped by pressing a non-space.

The command is another form of "tracing". Execution is begun at the target PC after a special breakpoint is set at the instruction physically following it. This is a shortcut way of temporarily setting a breakpoint and is usually used when the instruction at the PC is a BSR, JSR, TRAP #xx, TRAPV, TRAPcc, lineA, or lineF call and the user isn't interested in the workings of the routine, only in its result. It is also handy when the instruction is a DBcc and the user wants to let the loop continue and stop only after the loop is completed.

The ctrl-\ keypress also uses this special breakpoint for its "point and shoot" breakpoint (sets a breakpoint at the address of the disassembled line that contains the cursor and immediately starts execution). This causes no conflict with the commands use of this breakpoint since the ctrl-\ use takes place immediately and then it is done with it. If, however, the target program is currently executing when the ctrl-\ key is pressed, it automatically becomes a "jump in on the fly" breakpoint as discussed in the following paragraph (see also J command).

The J command's "jump in on the fly" breakpoint uses this special breakpoint also and its use does conflict with its normal use by the \ command. The J command removes any command breakpoint and replaces it with its own. This shouldn't be a problem, however, since the J command's purpose is to regain control of an executing target program which has managed to evade any set breakpoints; therefore, removing one of these ineffectual breakpoints shouldn't cause a problem.

The K command causes the target PC to be advanced past the instruction at its current value, thus skipping (not executing) that instruction. This is merely a shortcut way of changing the target PC.

Pressing those function keys types an underlined, backlit 1-5 or 0. The digit chosen builds the boolean tests for breakpoints 1 through 5 or the boolean test at each and every instruction (for the 0). The breakpoint is then only activated (counter decremented) if the test evaluates true.

The format is: digit: argument#1 relationship .size argument#2

The arguments can be one of the following:
pc (program counter)
sr (status register)
x[(x=0-7 for D0-D7)
x] (x=0-7 for A0-A7)
x[[(for D reg indirection)
x]] (for A reg indirection)
x[(x>=8 for contents of address x)
x[[(x>=8 for indirection with contents of address x)
x (for the constant x)

x is the usual free format and can be a 'symbol' instead.

Relationships include = , <> , < , > , <= , >= , and R.L .

Sizes include .B, .W, .L, and .M which are byte, word, longword, and bitmask respectively. Words and longwords do not have to be word aligned (they are built by fetching a byte at a time). Bitmasks have some special requirements; the first argument is either the SR status register or points to a longword (not necessarily word aligned) whose high word to be tested; the low word of the second argument is the mask for the first argument's high word and the high word of the second argument is the value to which the masking is compared.

Up to five relationship tests can be made for each breakpoint; the

relationships are combined with the boolean operators & (AND) or | (OR) . The tests can be grouped with parentheses to change the normal left to right evaluation. These grouping parentheses can NOT be nested (only a depth of one is allowed).

Each relationship test is entered on a separate line; the indication that this test is to add on to the current test rather than completely replace it is a & or | between the : and the 1st argument. Spaces are not allowed and cause a termination of the boolean build; therefore, a space after the : eliminates the boolean test (this is how boolean tests are cleared). Be careful that boolean tests aren't inadvertantly inherited; changing a breakpoint address probably means that any boolean test associated with the old breakpoint should be changed or cleared also.

examples: (the digits 3 and 2 which precede the following lines is actually the character that results from the F3 and F2 function keys)

```
3:4]>=.L3[[
3:|(pc<.L'start'
3:|pc>.L'end')
3:&('point'[=.W1234
3:|'point2'[<>.B'where'[[)
2:sr=.M00010005
2:&4[<>.M05000F00
```

The breakpoint is activated if the total test is true.

The first is true if the contents of the A4 register is greater or equal to the longword pointed at indirectly by the D3 register, or the program counter is either less than 'start' or greater than 'end', and either the word at the address 'point' equals 1234 or the byte at the address 'point2' does not equal the byte at the address pointed at

indirectly by the longword at the address 'where'. Whew!!! Those were all five of the possible tests allowable for that breakpoint number 3. The second is true if the status register carry bit is set while the zero bit is not and the D4 register has a value of x5xxxxxx (note the high word masking, it is x5xxxxxx not xxxxx5xx).

If more tests are desired, enter a new line (lead by a & or |) and press RETURN, it will be added to the end of the list. Editing or changing the order of the tests requires pressing RETURN selectively on each line starting with a line without a & or |, one RETURN for each test.

Thus far, the R.L relationship hasn't been discussed since it is a special case of the syntax; its format is either: address R.L address.length/ or address R.L address.end which are equivalent. The second block (at the second address) whose size is determined by the length/ or end is compared with the block at the first address. If any bytes in the blocks differ, the boolean condition is satisfied. This will be used mainly to trap when some write is made somewhere within a memory range; a carbon copy of the block is made somewhere and then the boolean test can detect changes. For speed purposes, the blocks are compared as longwords and limited to 64K; therefore, their addresses will be truncated down to an even address and length will be made a word which is a multiple of four.

Execution J

The J commands "jumps" into an executing target program and sets a special breakpoint "on the fly". Its one parameter is the address at which to set the breakpoint; it will "error" out if the target is not executing. Its primary use is to attempt to regain control of an executing target program which has evaded any other breakpoints set (probably by being in an infinite loop). If the target still doesn't stop executing, just move it somewhere else until it does. This breakpoint is shared with the \ command, which see. Note that this "jump in on the fly" breakpoint can NOT be set in the ROMs.

Note that this command only is available on a target program that was originally loaded as an executable file (with the > command) and then sent executing and is doing so currently (multitasking as a separate process). On the other hand, if the target program was created by hand assembling it (= command) in an allocated user memory block (U command) or perhaps loaded as a binary file, that target will NOT multitask; either KISSED will be running or it will be running but never both since they are in effect sharing the same task. Remember, when creating these special handcrafted target programs, allocate enough memory for a stack and point A7 at its end.

Note also the J command's breakpoint is the same one used by the \ command and the ctrl-\ keypress. Both the J command and the ctrl-\ keypress steal (remove and replace theirs) the breakpoint from command if needed. They also steal from each other; it belongs to whoever uses it last.

The N command creates, filters, and accesses the trace buffer. To use the other versions of the N command, the trace buffer must first be created with the sizeN command where size is the desired size in bytes. KISSED will allocate that memory for the buffer (and will automatically deallocate it when quitting KISSED). Note that the various filter options mentioned below lead to different things being saved in the trace buffer. Options 1, 2, and 3 all save only program counters; therefore, even a relatively small trace buffer of 10000 bytes ($2710) can save 2500 entries. Option 0, which saves all registers, actually judiciously saves only those registers that have changed values and thus, even with the masks and doubly linked list, averages only about 18 bytes per entry and thus has room for about 555 entries in that rather small buffer. If the buffer ever fills up, it wraps, overwriting the oldest entries.

A special case of the sizeN command, 0N is used to deallocate the trace buffer. Actually, if a trace buffer already exists, any sizeN command will kill the old one but will not create a new one. Therefore, to change the size of a previous trace buffer, first kill it with 0N and then create another with newsizeN .

The bare N command chooses the trace buffer filter by prompting the user to press a number key between 0 and 4, which represent progressively more restrictive filters. Each time KISSED regains control at least temporarily (periodically via a breakpoint being tested or every instruction via a trace, ROM breakpoint, or everytime boolean test), the filter determines what is saved in the buffer.

0 means that all registers are saved.
1 saves only the program counter
2 saves only those program counters that weren't in the ROMS ($00F80000 to $00FFFFFF)
3 saves only non-ROM program counters that are also the

calls to routines (subroutine calls, traps, lineA, and lineF calls)

4 is the ultimate restriction, it temporarily turns off the trace buffer.

A temporarily turned off trace buffer can be resumed again later by rechoosing the same filter as originally chosen (however, 1 and 2 and 3 can be freely substituted for each other). Incompatible original and new filters (when one is 0 and the other isn't) will empty the buffer before applying the new filter.

The N command followed by a string in double quotes accesses the trace buffer. Usually, this string will be the null string and the format of the command is N"" Initially, the most recent entry in the buffer is displayed. The left arrow key now can be used to move back to older entries; similarly, the right arrow moves toward more recent ones. The down arrow moves directly to the oldest entry while the up arrow goes to the newest. Note that if all registers are saved (option 0), the registers will be hilited as usual as their values change. This access of the trace buffer is ended by pressing the ESCape key.

If the string in N"string" is not empty, the command "searchs" the trace buffer for entries with instances of that string and stops only at those entries. This search is actually a literal, case-sensitive search of the displays that result from each entry and is typically used to search for addresses, data values, symbols, etc. The search is started from the most recent entry by pressing the RETURN key, new older matches will be found after each RETURN key press. When all entries have been searched back to the oldest, the search direction will reverse itself back to the newest. A search can be aborted at any time by a different key press. If that non-RETURN keypress is one of the four arrows or ESCape, they will behave as detailed above and ignore the string search aspect.

The ^ command is used to lock the register display in place. This command has a parameter of 0 or 1; 0 turns it off and 1 turns it on. The effect of this command is on the return from breakpoints or tracing. Normally, the display of breakpoints and registers occurs on new lines. However, when this command is on, the display occurs 6 lines back up the screen, thus overwriting the previous display of breakpoints and registers (if and only if the G, , or T command was on the line that immediately followed the previous display).

The purpose of this command is to protect other information even further up the screen from scrolling off. It also allows one key tracing since the T remains in the line to be executed again and again by a RETURN keypress. However, don't use this command just to obtain one key tracing; that option is also available directly from the trace (T) command.

It is advised to use this command rather sparingly, not because there is anything wrong with it, but rather because it limits the user's access to one of the the major strengths of the debugger. It is often very helpful to have a record of previous register values there on the screen.

The L command is used to disassemble memory into opcodes. There are four possible formats; no parameters, one parameter, two parameters, and finally "one and a half" parameters.

The normal format is start.linesL which disassembles a 2nd parameter word number of lines starting at the 1st parameter address (note that start.linesL is an exception to the usual start.end). If the 2nd parameter is omitted, a screen's worth of lines will be shown. If no parameters are given, a screen's worth will be shown starting from the end of the previous disassembly. The remaining format is start.L (Note that this format is somewhat strange in that the period says there is a 2nd parameter when there isn't one since the L command follows it immediately). The start.L version will continuously disassemble forever.

The output lines display the address (or symbol), the opcode (no extension words), the mnemonic and operands, and finally the address of the next opcode. Both invalid opcodes and INVALID ADDRESSING MODES for that opcode (unlike many other disassemblers) will be caught and won't disassemble (this is indicated by ** as the mnemonic).

Note that the ctrl-R or alt-R keys will provide the address at which to resume a listing.

Note also the effect of the alt-3 key which toggles processors

example:

23456.4L	Displays 4 lines starting at 00023456
*L	A very common command, displays a screens's worth from the current program counter.
*.16L	This has the same effect as *L if the screen has 23 lines (the line count is 22 decimal plus 1 blank line

that contains the cursor).

The L> command is used to disassemble the previously loaded target program to a disc file; it will generate labels and will thus be in a form nearly suitable for reassembly.

The L> command does its best to build a reassemblable source file. If it finds invalid opcodes, it tries to find the limits of these non-executable data sections and make them .dc.b's . It can, however, be fooled and thus some hand-editing will probably be needed for all but the simplest programs. The .dc.b lines are followed by their ASCII equivalents as comments.

Unfortunately, any lower case letters a thru o in these data sections will look like relative branches and spurious labels will undoubtably be generated by these. To aid in any hand-editing of these dc.b sections, the 42 bytes that precede and the 42 that follow the section are given before and after the section, respectively, as comment lines.

Longword constants that aren't addresses but happen to be in the range that could be will be made into labels and thus will have to be hand-edited back to constants. The labels created consist of an "L" followed by the address. Valid labels are produced by program counter relative addressing also.

Note the effect of the alt-3 key which toggles processors.

NOTE Large programs may take several minutes and several disc accesses; be patient, a completion message will be given. The file produced from a large program will itself be large (approximately 10 times larger); therefore, use a harddisc or start with a blank floppy.

example:

L>filename	Disassemble the previously loaded target program to a disc file of that name.
L>	Brings up the req.library file requester.

The = command is a one-line miniassembler used mainly for patches. It takes the standard mnemonics in the form used by the disassembler and converts them to code at some location. The format is x= (where x is the assembly address) followed by the mnemonics and operands. If no address is given, the assembly continues from any previous assembly.

To ease reassembly of disassembled code produced by the L command, the ctrl-N keypress is provided. When pressed in a disassembled line, it will delete the leading rubout and the opcode, place an = after the address and place the cursor there for editing. In addition, in subsequent lines, anything between a = in the first column and the mnemonic in the 18th column will be ignored. Reassembling a disassembly, therefore, consists of cursoring to the first line to change, pressing ctrl-N and editing the mnemonic and/or operands. Subsequent assemblies of the lines that follow need only to replace the rubout with an = and modify the mnemonic and operands if needed. Note that using the ctrl-N keypress in each line of a multiple line reassembly is a bad idea; if the previous assembly produced an instruction of different length than the instruction it replaced, the address starting this line will be meaningless (and since ctrl-N uses it, it will be in error while just an = in the first column ignores that address and correctly continues on from the previous assembly).

If the line is valid (valid mnemonic and operands), the line is disassembled on the same line with the ending address updated. If this ending address isn't the same as that required as the starting address of the next opcode, further patching or padding with NOP's will be needed. If the assembly fails, the first four columns will be hilited but the rest of the line will be unchanged so that the user can try to fix it again.

Note that to make KISSED more bullet-proof, it won't allow you

to assemble within KISSED itself.

Note also the effect of the alt-3 key which toggles processors.

If a target program being debugged, anything between the two single quotes is considered a symbol. The symbols are those that result when the target program was linked by the usual linkers with their include symbols option activated. KISSED uses those symbol hunks to build its own symbol table and it truncates any symbol longer than 32 characters.

This is a double use of the ' character, it is also used as a string delimiter in place of " when there is an imbedded " in the string. (see ' and S commands).

Symbols can be used anywhere a hexadecimal number is expected. If they are used in a fixed format line, they automatically override the fixed format (the ctrl-F isn't needed).

If a symbol isn't found, its ' will be hilited as an error. If a symbol is multiply defined, its ' and first character will be hilited. The symbols are also used by the disassembler and will replace addresses. For example, a line could look like this:

```
^'label'
^000EF012  7700 DBNE D2,$'loop' 000EEF32
```

instead of this

```
^000EF012  7700 DBNE d2,$000EEF32
```

Although the user can use symbols for any number, the disassembler will use them only for addresses and other longwords. To do otherwise would be to court confusion for example, just because some symbol has a value of 00000020, one wouldn't want every move immediate #$20 (putting a space somewhere) to be disassembled as #$'symbol'.

examples:

=BSR.W $'label'	Assemble that line
=MOVE.L D7,$'MORE'	Assemble that line
'another'.34/	Display that block of memory.

The ? command accesses the symbol table. A single parameter is an address and any labels associated with the address are then displayed.

For 3 parameters, the 1st determines the organization (0 = sorted by address, 1 = alphabetically) while the 2nd and 3rd limit the address range.

examples:

12345?	Display any or all symbols associated with that value.
0<'start'.'end'?	Display all symbols between the values of the symbols 'start' and 'end' sorted by value
1.FFFFFFFF?	Display all symbols sorted alphabetically

The * command merely is the current value of the program counter and can be used anywhere that a hexadecimal number is required.

example:

*L	Disassemble a screen's worth of opcodes starting from the current program counter.
*.1234+	Display the address offset 00001234 from the current program counter.

The H command does a hexadecimal search of memory block. It can search for either one, two, three, or four bytes (2, 4, 6, 8 digits) (symbols and { } math become 8 digit numbers). Each occurence within the block is displayed as the start of a fourteen byte group in both hexadecimal and ASCII so that the found bytes can be seen in context. Note that if searches longer than a longword are desired, the values can be made into a string and the ASCII search command (see S command) can be used.

Note that the display is in a form which can easily be edited for easy use as a command line.

Searches are made on byte boundaries.

Note that if the search ends, either normally or by aborting a scrolling display, the ctrl-R or alt-R keypresses will provide the address past the last 14 bytes displayed if the search is to later be resumed.

examples:

12345.12455H3456	Search the memory range for the two bytes 34 and 56
12345.111/H34	Search the memory range for the byte
13424.23/H'symbol'	Search the memory range for the 4 bytes of the longword value of 'symbol'

The S command is the search memory block command for a text string. It normally takes two parameters and thus the format is x.yS"string" where x.y (or x.y/) is the block to be searched and the " character delineates the string. If there is an imbedded " in the string, ' can be used instead.

Each occurrence within the block is displayed as the start of a fourteen character string in both hexadecimal and ASCII so that the found bytes can be seen in context.

If a third parameter is used, it is the hex value of a wild card character; it can be anything except whichever delimiter is used (' or ") or $00 (which is equivalent to no wild card).

Note that the display is in a form for easy reediting. Removing the rubout will allow the hexadecimal values to be changed by editing them and pressing RETURN. If editing the ASCII values is more convenient, remove the leading rubout, replace the enclosing two rubouts with either single quotes or double quotes (depending on the presence of the other type in the enclosed string), edit the string, and then press RETURN. Note also that if both hexadecimal and ASCII values are edited, the ASCII values will overwrite the hexadecimal changes because the ASCII changes are later in the line.

Note also that if the search ends, either normally or by aborting a scrolling display, the ctrl-R or alt-R keypresses will provide the address past the last 14 bytes displayed if the search is to later be resumed.

examples:

1234.1332S"Hello"	Search that range.
1234.FF/S'say "Hello for me"'	Search that range.

3F<123.FF/S"a?c?"	Search that range for four characters, a and c each followed by anything. Note that the $3F specifies the ? character as the wildcard.

The V command is the verify memory block command. It takes three parameters and thus its format is z<x.yV (or z<x.y/V)

It compares the x.y block with a second starting at z and hilites the differences in bytes, regardless of the memory width. Each differing byte also has the eight bytes following it displayed so that it can be seen in context; the byte in question is thus the first in the string of nine bytes. The display is in a form for easy reediting; just remove the rubout, make the changes, and then press RETURN.

Note that if the verification display ends, either normally or by aborting a scrolling display, the ctrl-R or alt-R keypresses will provide the address past the last 14 bytes displayed if the verification is to later be resumed.

examples:

12345<23333.24333V	Compare the block from 00023333 through 00024333 to the block starting at 00012345.
12345<23333.1001/V	Same as above command

The M command is the move memory block intelligently command and takes three parameters and thus has a format of z<x.yM

The block starting at x and ending at y (inclusive) is moved to a new location z (as usual, y can be replaced by a length by making it y/). If the old and new blocks overlap, the copying is intelligently done so the new block is correct (copying from top down or bottom up as needed).

12345<6789A.678FFM	Copy the block between 6789A and 678FF to the location 12345.
12345<6789A.66/M	Identical with above command

The M+ command is the copy memory block forwards in memory command. It takes three parameters and thus has a format of z<x.yM+ (or z<x.y/M+). The command copies forwards in memory, from the bottom up. Note that block moves are done automatically with the M command, but sometimes one wants to force the direction of copying (usually, this is done to make repeating patterns by overlapping the blocks).
examples:

12345<6789A.678FF+	Copy the block between 6789A and 678FF to the location 12345.
12345<6789A.66/+	Identical with above command

The M- command is the copy memory block backwards in memory command. It takes three parameters and thus has a format of z<x.yM- (or z<x.y/M-). The command copies backwards in memory, from the top down. Note that block moves are done automatically with the M command, but sometimes one wants to force the direction of copying (usually, this is done to make repeating patterns by overlapping the blocks).

Note that to make KISSED more bullet-proof, it won't allow you to overwrite KISSED itself.

examples:

12345<6789A.678FF-	Copy the block between 6789A and 678FF to the location 12345.
12345<6789A.66/-	Identical with above command

The Z command is the fill memory block with byte constant command. It has three parameters so the format is z<x.yZ (or z<x.y/Z) which fills the x.y block with the byte z.

Note that to make KISSED more bullet-proof, it won't allow you to overwrite KISSED itself.

examples:

0<12345.12446Z	Fill the block with 00's
AF<12345.102/Z	Fill the block with AF's

There are three uses of the > command, which deals with disc files, depending on how many parameters are passed to it. In each use, the filename which immediately follows the > must be enclosed in double quotes (") if it contains any spaces. If no filename follows the >, the req.library file requester will appear.

examples:

>target	loads executable program
>"has spaces"	"
7CF5000>filename	loads file at that address
0>"another file"	allocates memory and loads file
7C45000.200/>filename.xxx	write file
7C45000.7C451FF>"has spaces"	"

If one parameter is passed to the > command, it is an address and the file name following the > (enclosed in double quotes if it contains spaces) will be loaded at that address. KISSED makes no judgements and assumes the user knows what he/she is doing and will attempt to load files of any length anywhere desired. KISSED will, however, protect itself from being overwritten and will display an appropriate error message when this is attempted. The file will be loaded as is; if executable program files are loaded like this, the load will merely be a raw binary load.

As a special case, if the "address" is zero (ie 0>filename), KISSED will use one of the user memory blocks (see U command) to allocate memory enough for the file.

If two parameters are passed, there are start.end or start.length/ of a file to be written to disc;the user is then informed whether or not it was successfully written.

This command is not intended to allow the writing of patched

executable files, a very bad idea. They wouldn't have their relocation information, etc unless loaded by address>filename and then, unless the patch was trivial and didn't change either length or any addresses, the relocation information would now be wrong.

If there is none (no number preceding the >), the target file whose name follows the > is loaded into memory along with any symbol table. If the load was successful, the address of the task structure is displayed followed by the start and length of each of the segments in the scatter load. Finally, the starting registers and instruction are displayed. The target program is now ready for execution or tracing (probably after some breakpoints are set) as a separate process in the multitasking environment.

If an attempt is made to load another target program after a previous one was loaded (and hasn't terminated), the user will be asked what to do with the old one. If the old target is currently executing, the options are abort the new load or let the old target run and do the new load. Aborting the new load ends the command.
Letting the old target run means that KISSED will break its ties to the old target (remove breakpoints, toss symbol table, etc) and then will load the new target program. If the old target is in a good enough shape, the optimum solution is to not attempt a new load until the previous target has terminated itself through its normal cleanup procedure. When this has occurred, KISSED will be notified, will toss the now useless symbol table, etc and will then notify the user with a "task terminated" message. Now the new load can process with a cleaner system. There is one potential problem that may occasionally occur when letting a target program run; it occurs when the target program changes its TRAPCODE, the trap (exception) handler for that task. See the & command for a complete discussion of this problem and its solution.

On the other hand, if the old target program is currently stopped, the options are to abort the new load, remove the old target and do the new load, or suspend the old target and do the new load. Unfortunately, removing the old target will probably be a very incomplete procedure since the state of resources, libraries, extra memory allocations will be unknown by KISSED. Therefore, if it is capable of running, start it up again and let it terminate itself (perhaps by repointing the program counter at its cleanup code) and continue as discussed above. Another less satisfactory but probably safer solution is choose the "suspend" option. This option merely patches the target by overwriting 14 bytes at the current program counter. This patch merely calls the Exec Wait command with a null signal mask so that it will wait forever. This won't free up any of the memory, etc but will put the old target task in a safer known state in which it can safely be set free to "run" (actually to wait) and a new target loaded. Be sure that this 14 byte patch won't overwrite something still needed even by a suspended target program (such as an interrupt handler, etc).

The ! command loads and executes a user's file of KISSED debugging commands. First the 0!filename (or 0!"file name with spaces") command is used to load the debugging file. The memory for the file is allocating by temporarily using one of the user memory blocks (see U command); this block is released when the whole file has been executed.

The command lines in the file MUST end with at least two spaces and be less than 80 characters in length, including those two spaces. The line terminator is the usual linefeed, ASCII $0A.

After the file is loaded, the 1! or ! commands are used to execute the file. The 1! command single steps through the file a line at a time. The user executes a line by pressing RETURN will the cursor is in the line; the 1! that KISSED appends to each line automatically recursively brings up the next line. The ! command executes all lines in the file without having to press RETURN after the first time. Note that spacing out the 1 in the 1! appended to a single step line will cause the whole rest of the file to autoexec.

Note that several commands, particularly those which require user interaction, end the parse of the command line that they are in, thus stopping before the 1! or ! that ends the line and produces the recursion. To restart the process, merely delete the line through the command last executed and press return to continue. As another example, if the command is a = command to invoke the mini-assembler, the ! will again never be reached, therefore the user should merely start a new command line with a ! or 1! and a return to restart the debugging file.

Creative use of autoexec debugging files can be very timesaving. For example, to set 4 of the 5 breakpoints in an already loaded target program, fill some memory, load some other memory, and finally disassemble 8 lines make the following text file:

```
b: 1.'part1' 1.'theend' 30.'loop23' 1.'never' 0.0
FF<'table'.A0/Z 4W 'table';'theend' X 'loader'.'part1' 'where'.8L
```

examples:

0!filename	Loads that file of commands after allocating memory for it
1!	Executes a single line of that file. Since KISSED adds a 1! to the end of subsequent lines, pressing return will continually execute one more line.
!	Execute all the lines immediately.

The & commands deals with the stack size used when creating a process when a target program is loaded (the default is 4000 bytes, $FA0). A bare & will display the current size that will be used while size& will change that size.

The bare & command also displays the default TRAPCODE address which may be needed when deciding to let a target program run when either quitting KISSED or loading a replacement target program. When KISSED first loads a target program, it "steals" the TRAPCODE vector so that it can both trap the usual exceptions (bus error, address error, division by zero, traps, etc) and do breakpoints and traces. If a target program includes its own TRAPCODE handler, it will steal the already stolen vector from KISSED to do its own handling. As long as it is well behaved and passes on to KISSED the illegal and trace exceptions (for breakpoints and traces, respectively), KISSED is happy and functional even though it is getting the exceptions secondhand. Target programs should still be happy since they shouldn't need those two anyway. The problem comes when KISSED is to release control of the target (quitting or a different target load); part of that release of control is replacing the default TRAPCODE handler since the target is now on its own. KISSED, however, is smart enough to recognize its own handler and won't replace the default unless what it replaces is the KISSED handler (if it did, the target's handler would be bypassed). The target program, on the other hand, knew nothing of KISSED and assumed that any exceptions its handler couldn't handle were going to the default handler (not knowing it had previous been stolen). Now, with KISSED going away and taking its handler with it, any exceptions passed on by the target's handler will go to never-never land. Therefore, before letting a target with its own trap handler run on its own, examine its trap handler and patch it so that the default address (to which it passes all exceptions that it doesn't want) is the real default vector and not KISSED's vector.

This bare & command furnishes that real default vector.

The # command takes one parameter and converts it from a hexadecimal number to decimal and displays the result.

The $ command takes one parameter and converts it from a decimal number to hexadecimal and displays the result.

Note the difference between the $ command and the % command; the $ command merely displays, the % command allows the result to be used as a parameter.

examples:

FE23#	Displays 65059, its decimal equivalent.
1000000#	Displays 16777216
65059$	Displays 0000FE23

The # and $ number conversions do only conversions and display of results. The command 50000<60000.100$/M will **NOT** convert the 100 from decimal to $64 hex and then move 100 ($64) bytes of memory from 60000 to 50000. The $ (and #) command is a separate command in itself; the above command sequence would cause the 100 to be converted to $64 and displayed (the extra 50000 and 60000 parameters would be thrown away since the $ command doesn't need them). Next the command line would be echoed with the / command error flagged since the / command needs two parameters but has none since the $ command stole them. Instead, the % command should be used, 50000<60000.100%/M will move one hundred bytes.

The (command provides floating point decimal input. The decimal number is enclosed between a (and a). The result is a floating point longword in the form,
mmmmmmmm mmmmmmmm mmmmmmmm seeeeeee
That is a 24 bit Mantissa, a Sign bit, and a 7 bit Exponent (64 or $40 offset). This number can be used like all other numbers but is undoubtably only useful in math operations.

examples:

(1234.56).(-.00345)~*F	Multiplies those two numbers and displays the hexadecimal longword result.
(.125).DCCCCD42+F	Adds .125 to 3.45 and displays the hexadecimal longword result.

Note - Using the (number) command by itself, not a part of a math operation, is definitely dangerous; KISSED will it as a number which will be greater than $80000000 (or 0) and, since no command follows it, its command will be the space command which displays memory. Therefore, the byte, word, or longword (depending on the current memory width) at that very high address will be displayed. On some system configurations that memory is mapped strangely and this reading of a location up there could make the hardware do strange things. If the user merely wants to see the hexadecimal representation of (number), making it part of a "do nothing" math operation such as (number).0+ will display the result without using that result as a memory display address.

The) command displays a hexadecimal floating point longword as a formatted decimal floating point number. The format is FFFFFFFF DDDDDDDD.number)
where F is the total field width (from 1 to 13 or 1 to $0D hexadecimal) and D is the number of decimal places. If a number is too large so that the number of decimal places plus the number of places needed to the left of the decimal point plus room for the decimal point and maybe a minus sign exceeds the stated field size, the number of decimal places is reduced to still fit into the field. If the number is even larger so that even the digits to the left of the decimal point won't fit, the lowest order of them are lost.

Note that if the) is paired with a matching (, this is not this decimal output command but is instead the decimal input command which was discussed just previously.

examples:

804.F7800047)	The number 123.75 will be display as 123.7500 (4 decimal places in a field of 8)
0704.F7800047)	The number 123.75 will be display as 123.750 (4 decimal places in a field of 7). Note that one decimal place was lost to fit it into the field; if the number had been negative, still another decimal place would have been lost to make room for the minus sign.
D07.(654.321))	yields 654.3210500 which shows some roundoff error in the eighth significant figure. The floating point calculations are done to about 7 decimal digits of accuracy with some small potential roundoff error in the seventh digit. The 24 bit mantissa makes this inevitable since 24*log(2) ~= 7.2

The + command merely adds the 1st parameter to the 2nd and displays the result. If the result of the addition is needed as part of some other command, it must be enclosed in { }.

The - command merely subtracts the 1st parameter to the 2nd and displays the result. If the result of the subtraction is needed as part of some other command, it must be enclosed in { }.

The ~ command does hexadecimal math as determined by the operator following. Therefore, 12345678.9ABCDEF0~+ is the same as 12345678.9ABCDEF0+ ; also ~- is the same as - .

1234.5678~* displays the product of words 1234 and 5678.

1234.56789~/ is in the format divisor.dividend~/ and displays the remainder and quotient of the word 1234 divided into the longword 56789 (0019004C , the remainder is the high word). Division by zero or quotient overflow results in a hilited error display.

If an S follows a ~* or ~/ , signed math will result.

If an F follows a +, -, ~+, ~-, ~*, or ~/, floating point math will result.

~< and ~> are the shift operators in the format value.shiftcount~< while ~&, ~|, ~^ do AND, OR, EOR respectively.

There are to other operators, ~~ and ~% , which are NOT and a binary display, respectively. Note that these last two operators only take one parameter.

Additionally note that only +, -, ~+ , ~- , ~* , ~/, ~+F, ~-F, ~*F, ~/F, ~< ,and ~> can be enclosed in { } for embedded math (no ~*S, ~/S, ~~, nor ~%). Care should be taken with { } enclosing a

~/ since a remainder in the highword will probably give undesired results.

Finally, note that the result of mathematical operations can also be accessed with the ctrl-R and alt-R keypresses.

examples:

12345.26FF+	Adds and displays the result.
{12345.26FF+}.30/	Adds 00012345 to 000026FF and displays 48 bytes from that address.
12345.26FF-	Subtracts 00012345 from 000026FF and displays the result (it will be negative).
14.3456~*	Multiply and display the result.
FFDE.3456~/S	Signed divide FFDE into 00003456 and display the result.
89ABCD43.ABCDEF51~/F	floating point 87963.87/4.302222 = 9FBC4C4F or 20446.15

The delayl command changes the scrolling rate. The default is the fastest rate and is equivalent to 0l . Delays are in increments of 0.02 (one fiftieth) of a second and occur between scrolling lines.

A scrolling display can be paused by the space bar, then restarted by pressing it again. RETURN or any other non-printing keypress will scroll the display a "page" at a time; any other keypress during a scrolling display aborts the display.

Note also that the right mouse button can be held down to pause a scrolling display.

The ‘ command displays and/or changes the colors in KISSED's palette. A bare ‘ command displays the four colors in a format of RGB.colornumber‘ which is itself the format for changing one color. Therefore, if the leading rubout and "palette" text is deleted, the various RGB values can be edited and then executed.

The _ command allows access to the task structure of the target program. A bare _ causes the display of the address of the structure and also the address and length of the various segments.

If the _ is followed by something other than a space, the structure address can be used just like any other number.

Two consecutive _'s, __ , is used to display the various lists associated with ExecBase. There are nine lists to choose from: MemList, ResourceList, DeviceList, IntrList, LibList, PortList, TaskReadyList, TaskWaitList, and SemaphoreList. Whichever list is chosen, the address of each node will be displayed followed by its "name". See also Warnings and Tips #14 for a flexible way to manually walk through lists.

The P command toggles the printer on/off. KISSED's output is redirected to the printer (as well as the screen) by the P command. It is turned off again by another P on that or some other eventual command line.

Commands which may send non-printable characters (00-1F, 7F-9F, and FF) to the printer will obey the following rules in converting from the screen's representation to the printer's.

Characters that are backlit hexadecimal digits (00-1F, 80-9F) are italized hexadecimal digits instead
Characters that are underlined (10-1F, 90-9F)remain underlined
Characters that are hilited in red (80-FF) are in boldface instead
7F becomes an underlined X, FF an underlined boldface X.

The PM command toggles "print" to memory on/off. Printing to memory redirects KISSED's output to a user designated memory buffer (as well as to the screen). It is turned off again by another PM on that or some other eventual command line. The memory buffer will usually be one of the user memories previously allocated. A typical sequence, therefore, will be:

1000.0u4 u4.1000/PM 14560.40/ 'table'.100/ PM

First, some memory (4K) is allocated. Then, that memory's start and length is supplied to the start.endPM command which starts the print to memory. All subsequent output will now be placed in that buffer; in the above example, this is two separate displays of memory. Finally, another PM ends the printing to memory. Of course, all of the commands above didn't have to be on the same line. When the ending PM is executed, KISSED will display the start.length/ of the memory actually used. Note that if a filename is now tacked on the end of that start.length/ , the used portion of the buffer will now be written to disc; this is probably the usual use

of the printing to memory command. The resulting file can then be used in the user's text editor if desired.

The Q command quits KISSED and reenters the desktop or CLI. Before quitting, any allocation of user memory blocks (unless they were made permanent via the 1Ux command which see) are freed up automatically. If quitting is attempted with a target program loaded (executing or stopped), the user will be presented with the same options that were presented when the user tries to load a second target program when one had been previously loaded. See the >filename command for a discussion of these options and their consequences.

If clones are being used, a Q command from within a clone will do the procedure discussed in the above paragraph and then access the original KISSED (clone #0). On the other hand, a Q command from within the original KISSED (clone #0) will not be executed if any clones are still active (not Quit). Thus, only clone #0 can quit to the desktop and it can do this only when any other clones have themselves been quit.

The >> command creates and/or accesses clones of KISSED to allow multiple tasks and/or processes to be controlled simultaneously. Of course, separate processes could be controlled by merely running multiple copies of KISSED but the >> command provides a memory saving alternative. Each separate copy of KISSED would take over 100K (for the 48 line version) each; using clones, the first copy of KISSED takes the 100K but each additional "copy" (clone) only takes an additional 5K for a considerable savings. In addition, separate copies of KISSED would provide no way of controlling "child" processes spawned by the original process; cloning provides the "hooks" to control the children or even grandchildren, etc. A maximum of 10 clones are available, each with their own breakpoints, user memory/variables, trace buffer, character screens (but they share the bitmap screen), etc. The clones can be differentiated by their clone number (0-9) in the screen title and their background color which is different for each clone.

The >> command takes three forms. A bare >> creates a new clone; the > command can then be used as usual to load in a process for that clone to control. A >>x where x is a digit between 0 and 9 accesses one of the 10 clones; it will error out if that clone doesn't exist. Accessing a clone first saves the current clone's screen, cursor, etc and then retrieves and displays that of the newly requested clone. If the user is currently interacting with some clone and a different clone (controlling some other process) hits a breakpoint or has its process terminate, that different clone is automatically accessed (this will be deferred temporarily if the user currently has a "popup" displayed).

The third form of the >> command is used to control child processes (created by CreateProc) or child task (created by AddTask). Once either of those two functions is called, they're off and running and it is too late to control the new child. Therefore, if

execution of the parent process is stopped at the AddTask function call, a bare >> will now automatically create a new clone which puts its "hooks" into the child, adds the task, and immediately stops the child at its first instruction. Meanwhile, the parent process's PC is incremented over the AddTask function call (JSR d6(a6) or JMP d16(a6)) and its D0 reflects the result of the call. Similarly, if the parent is stopped at a CreateProc call, a >> command will then create the new process, etc just like was done for AddTask; however, for CreateProc, the >> command is not bare but is instead >>"filename". The quotes are needed even if the filename contains no spaces. The filename is the file which will be read to obtain the symbols and will be the same as that provided by the parent for the LoadSeg command that will preceed the CreateProc.

Note that a clone can NOT set breakpoints in a process that it doesn't control; it has no "hooks" there and the breakpoints will appear as illegal instructions.

1. Memory locations whose address must be even (word boundaries) such as the values for the PC, memory change and display addresses when the width is word or long, assembly point for the miniassembler, file read address, etc will be forced to be even by KISSED changing their least significant bit to zero. This forcing to even, of course, is done only when within KISSED, if the target program somehow changes the PC to an odd number, an address error will result. Stacks aren't forced even by KISSED. When displaying words or longwords with the space command, any forcing to even will be indicated by hiliting the addresses in red.

2. Patching with disassembler-assembler combination can be a very useful debugging tool. Since editing a source file, assembling it, and relinking the result can be a lengthy process, it is often advantageous to temporarily patch any bugs found so the debugging process can continue to find more bugs until enough have been found to make using the editing- assembly-linking process worthwhile. These patches are not meant to be permanent and KISSED reinforces this by making no provision for re-writing files, patched or otherwise. However, temporary patches are made easily. If the patch will fit into the original code, simply disassemble some code, point the assembler and place the ='s to reassemble it.

A more typical patch session where the patch won't fit inline might go like this. First, the area around the bug is disassembled. Next, a user memory block is allocated using the U command and the assembler is pointed at its start, the disassembled code is edited, changed, and reassembled creating the patch. Then, the code is disassembled again and the jump to the patch (and any necessary NOP padding) is assembled. Note that this procedure preserves any relative branch address destinations in the patch; since the disassembly specified the destination addresses and not the offsets,

KISSED automatically builds the new reassembled offsets. The offset size (short or word) is specified by .S and .W ; if neither is specified, KISSED will use the short offset if the address is within range or else will use a word. If even the word offset +/- 32K isn't enough, then a JSR or JMP will have to be used. If patching is being done, be careful when changing offset sizes; the .S opcode instruction fits in two bytes and the .W in four while JSR's and JMP's take six. For the 68030, .L offsets are also available.

3. If the P command is used to toggle the printer on/off (output directed to printer and the screen), but will be ignored if the printer won't accept characters (it isn't on, out of paper, etc).

4. The miniassembler expects mnemonics in the primary form (which is also the form produced by the disassembler); alternate mnemonics are not supported. For example, ADD.W #$1234,D3 is the correct form, the alternate form of ADDI.W #$1234,D3 will not assemble. Similarly, ADD #$1234,D3 will not assemble, the alternate form with implied word length is not allowed. Also, ADD.L A0,A3 is correct while ADDA.L A0,A3 is not. Numbers are hexadecimal and need their $; however, in accordance with the standard practice, the $ is not used for the 3 bit immediate counts. Thus, LSR.B #3,D2 and ADDQ.L #4,D3 are correct. Also, in accordance to standard practice, TRAP #14 and not TRAP #$E is correct. The mini-assembler has one quirk, it can't assemble correctly MOVEM.L or MOVEM.W mnemonics that use only single registers. An example of this is MOVEM.L D3,-(A7) . Normally, MOVE.L D3,-(A7) would be used anyway; however this affects condition codes and MOVEM doesn't. If single registers are wanted, either of the following will force the correct assembly, MOVEM.L D3-D3,-(A7) or MOVEM.L D3/D3,-(A7) .

5. Tracing through breakpoints decrements their counters normally until they try to decrement to zero, when this occurs KISSED

assumes the user knows what he/she is doing and continues to set them back to one. Note that this means that if two breakpoints are set in the same loop but with different count values, the larger count will continue decrementing when the smaller remains stuck at one. There is one other clarification about breakpoint counters. Unlike some other debuggers which don't allow execution to resume when the PC is at a breakpoint, but force the user to trace past the breakpoint before execution can be resumed with the breakpoint placed, KISSED handles this automatically. When the target program resumes execution when its PC is at a breakpoint whose count is one, KISSED assumes that the user wants to stop execution when the program returns once again to the location, not immediately (since the PC is already at the breakpoint). Similarly, a count of two will result in the breakpoint being activated the second time around, not counting the current presence at the breakpoint, etc. To achieve this, KISSED doesn't insert the breakpoint immediately (it can't since it can't execute the breakpoint itself) but instead traces for one instruction and then inserts the break-point where it belongs. this is all transparent to the user and is only mentioned here in case the user was wondering how it was done. It does mean that there is a multiple use for the trace bit, normal user tracing with the T command and this special tracing unnoticed by the user.

6. "Popups" (which result from the help and ctrl-V keypresses as well as the window into memory snapshots, etc) while being shown, defer communication from the target (breakpoint returns, target termination, etc) and so should not be left in a "popped up" state indefinitely. When removed, deferred communications resume.

7. Because of KISSED's use of the trace bit, KISSED will try to prevent the user from using it. Setting it by making its bit 1 in the change register command will be ignored. Tracing through an

instruction which sets it (the target program would have to be in supervisor mode) is software simulated to prevent it from being set and causing KISSED to lose control of the trace (see T command). If the user forestalls all of this protection and has a target program set the trace bit during normal execution, the Amiga will GURU. Don't try to trace into exceptions (or set breakpoints there) since a GURU will result from any excection that occurs in supervisor mode.

8. Multiple pass breakpoints are very useful, but there is a penalty involved also. Every time a breakpoint is passed through and its counter decremented, several thousand machine cycles are needed by KISSED for processing that and the other breakpoints, etc. This is normally not noticeable; however, if the breakpoint is set with a very large count in a loop where the several thousand cycles will significantly lengthen the processing time of the loop, the execution of the target program can be noticeably slowed. This effect can be used to the user's advantage if slow motion execution is desired.

9. The ^ command is handy to keep previously displayed information from being scrolled up off the screen during multiple traces and/or breakpoint returns. It also allows the G and T command line to be executed repetitively by merely pressing RETURN.

10. ctrl-C, the copy line keypress merely makes a duplicated of thecursor line in the line below it. An example of its usefulness is as a template. Suppose you want to modify some displayed line of memory, strings, registers, or whatever. Simply display the current line of values, cursor up and ctrl-C, you then can modify the new line while the old one is immediately above it for reference. This has the added advantage that cursoring up to the old line and pressing RETURN will undo the changes.

11. The ctrl-W keypress is a convenience to be used when the cursor is in a disassembled line of code containing either symbols, pc relative addressing, ie. move.l $1234(pc),d0 , or other hex number preceded by a $. Pressing ctrl-W when in such a line will open a blank line. For each symbol, its associated longword is displayed as will the address or symbol pointed at by the pc relative address. Other numbers become ASCII. Pressing ctrl-W in the breakpoint line displays the symbols for those breakpoints.

12. Breakpoint boolean tests on words or bytes that use memory indirection need some care in their use. All values fetched from memory are fetched as longwords and then the test is made. Therefore, if a word sized test is made and one of the arguments is 70000], the word tested is 70002; if the word at 70000 is needed, the argument would have to be 6FFFE]. Similarly, if the test was byte sized, the byte would be at 70003; to get the byte at 70000, the argument would have to be 6FFFD (this is legal since the longword fetched is built up byte by byte.

13. KISSED needs the illegal and trace exceptions to operate (for breakpoints and tracing). They are passed to KISSED through the TRAPCODE vector of the target program. If the target program steals the vector and doesn't pass them through to KISSED, KISSED can do nothing about it. Normally, however, target programs don't want those exceptions so they pass them on to the default handler; in this case, however, KISSED has stolen that vector so gets them instead. See further discussion in the >, Q, and & commands.

14. The indirection command is very useful and powerful. As an example of both, let's assume that the A6 register contains a ptr to ExecBase and you want to see what tasks are in Exec's task waiting list. The following two lines will browse through the list.

{1A6.6]+}]U3
{U3.A+}]O U3]U3

First, 6] accesses the contents of the A6 register (ExecBase) and 1A6 is added to it to point at the TaskWait pointer. The] indirection command fetches that pointer and it is placed in the user variable U3 (the { } brackets provide embedded math which allows the result of one math operation to become an operand in another). U3 now contains the address of the first task. Second, the task's name string ptr is offset at 10 ($A) from the task structure's start; the] command fetches the string pointer and the O command displays the string. The other command on the second line links from one task to another; the link pointed at by U3 is indirectly fetched and replaces the current value in U3. Finally, repetitively cursoring up to the second line and pressing RETURN, recursively displays each task in turn. As a refinement, replacing the second line with

{U3.A+}]O U3U2 U3]U3

saves the displayed task's structure pointer in U2 so that it is available for use if the user is looking for a specific task by name (otherwise, U3 has already been linked to the next task).

15. There are some things to watch out for when using the L> disassemble to disc command. The first, and least significant, is replaceable opcodes. For example, the 68000 has two distinct instructions, ADDI.B #$12,D4 and ADD.B #$12,D4 . ADDI is specifically and only add immediate while the ADD instruction shown is using the add immediate addressing mode of the many addressing modes it has available. The effect of these two instructions are indistinguishable and assemblers use them interchangably. KISSED disassembles both to the more general ADD.B mnemonic; it uses no alternate mnemonics (as discussed in

tip #4 above). This means that if the original file used the ADDI form, KISSED's disassembly would use the ADD form; if the disassembly was then reassembled, the ADD form would be used and the original and subsequent object codes would differ (although they would execute identically).

If modifications are made to the disassembled listing before reassembly, beware label use hidden in data structures. On the surface, it would seem that changes and/or additions can be made with impunity since all absolute addresses have been changed to labels and therefore any changes in length are compensated for automatically. On closer reflection, however, labels can be hidden elsewhere. For example, consider the following code fragment which uses the 0-9 value in d3 to choose between 10 functions to execute:

```
lsl.w   #2,d3                ;four byte (longword) entries
move.l table(pc,d3.w),a0     ;fetch chosen function address
jmp     (a0)                 ; and do it
table:
dc.l    function0            ;on the 68030, all 3 code lines could
dc.l    function1            ; could be replaced with a
. . .                        ; jmp ([table,pc,d3.w*4],)
dc.l    function9
```

KISSED will realize that the table isn't executable code and will disassemble it as raw dc.b's . However, if the code size changes and thus the label values change, those raw dc.b's don't change to match, KISSED thinks that they are data. The user will have to recognize the special use of that data structure and change those dc.b's to the form, dc.l label. The following code fragment which serves the same purpose is even harder to find and fix.

```
lsl.w       #1,d3               ;two byte (word) entries
move.w      table(pc,d3.w),d3   ;fetch the chosen offset
jmp         table(pc,d3.w)      ; and use it
table;                          ;note that the base address
dc.w        function0-table     ; of the offset and table
dc.w        function1-table     ; could be different, ie
...                             ; jmp  other(pc,d3.w)
dc.w        function9-table     ; dc.l  function9-other
```

16. The addrGH "go hook" command has some potential problems, the main one being the "eating one's own tail" problem mentioned in the discussion of the command. That problem involves stopping that hooked process with KISSED while simultaneously attempting to use that process elsewhere; a prime example of the perils of multi-tasking. For example, suppose the user sets a breakpoint in the process that controls reading a disc drive and then attempts to use that disc drive from within KISSED via this or some other clone. If the user persists and continues on in the routine that was called to use the disc and finally to where the routine was called, the user could end up setting breakpoints within KISSED itself which will almost certain lead to disaster. Fortunately, KISSED uses very few system resources unless as result of the user's direct action, such as reading a disc; only the console.device is used continually. The operating system's resource management procedures also should provide some protection.

There is another potential problem with using the "go hook" command to control an already running process. To control a process, KISSED steals the task's trapcode vector; if the running process has previously redirected it itself for its own purposes, KISSED stealing it will undoubtably cause problems. In the normal procedures of controlling a process as it is created, using the >> command rather than the GH command, KISSED steals the

redirection works as desired and, as long as the illegal instruction and trace exceptions are still passed on to KISSED, KISSED works too. There is no good solution for this problem. Fortunately, few processes redirect their trapcode vector and none of the usual device processes do and these will be the usual targets of the "go hook" command (other processes will normally be controlled from the their inception).

17. To control a process or task, KISSED does two things to it. One is the stealing of its trapcode vector mentioned above and elsewhere. The other is that it uses one of its free signals for communications between it and KISSED. When KISSED is used to create the process, it gets first crack at the available signals so it always gets one; as long as the process doesn't need all of them including the one KISSED took, there is no problem. The "go hook" command, however, tries to get a signal after the process is already running; if none are left, KISSED can't take control.

18. If it is necessary to pass WBArgs to a target program (If, for example, the target program is to be used by using extended select to select data files for the target program before double clicking on the target program icon), then pretend that KISSED is the target program. By this I mean, do the extended select on the desired files and double click on the KISSED icon. When the target program is loaded with the > command, KISSED will pass the ArgList on to the target program to be accessed as usual through their WBStartup message.

19. The user can change the help text file S:KISSED.HLP (or even kill by deletion or renaming). The entry for each command character starts with that character and a linefeed. Then comes the text, at most 79 characters per line, ending with a linefeed. Entries can be in any order and aren't necessary for all commands. Two bare linefeeds end the entries.

<	start of 2nd block indicator (or fill value for Z command)
%	consider number to be decimal rather than hexadecimal
.	start.end or start.length/ separator (. and , are equal)
,	start.end or start.length/ separator (. and , are equal)
/	the normal start.end becomes start.length/ instead
{ }	use imbedded math
O	output ASCII memory strings
:	enter hexadecimal bytes into memory regardless of width
;	enter hexadecimal values of current width into memory
space	display memory block
"	enter text into memory (if imbedded " , use ' instead)
'	enter text into memory (if imbedded ' , use " instead)
W	change memory width
X	end of change memory values
@	create/remove "window into memory"
U	allocate user memory blocks or set variables
I	inspect target breakpoints and counters
b	change target breakpoints and counter
R	display target registers
p	change target PC, Usp, SSP, and SR
a	change target Address registers
d	change target Data registers
Y	use old register values
[ ]	memory indirection by A or D registers or memory pointers
G	Go execute
GG	Go til logic fork disturbs sequential execution
GH	Go hook uncontrolled process
G]	Go until reach address on the stack top
J	set "jump in on the fly" breakpoint
K	skip over opcode at PC
N	create/filter/access trace buffer
F1-F5	Build boolean test for breakpoints
F10	Build boolean test for always

T	trace target program, numberT traces continuously
\	execute until breakpoint immediately following PC opcode
^	overwrite previous register display upon return
L	disassemble opcodes
L>	disassemble target program to reassemblable disc file
=	assemble opcodes
?	search symbol table for address or display symbol table
'	use value from 'symbol' table as number (see " above)
*	use target PC value as number
H	search memory block for hex values
S	search memory block for ASCII string
M	move memory block automatically
M+	move memory block forward
M-	move memory block backward
V	verify memory blocks
Z	fill memory block
>	load target or read/write files
!	load/execute debugging command file
#	display as decimal/hexadecimal
(	input floating point decimal
)	display floating point decimal
&	set/display target task stack size
|	change scrolling speed
+ -	hexadecimal add/subtract (end +/- start)
~	hexadecimal math using operator following
`	display or change KISSED color palette
P	toggle printer
PM	toggle "print" to memory
Q	quit KISSED
_	display/use target's segments, task address
__	display address of various operation system lists

Default Screen Editing Keys

left	move cursor, wrap screen
right	
up	
down	
tab	tab to next hexadecimal value
backspace	backwards character delete
del	forwards delete
ctrl-A	go to start of line
ctrl-C	copy current line
ctrl-D	delete through the next rubout
ctrl-E	go to past last non-space in line
ctrl-F	override fixed format
ctrl-K	delete to line end
ctrl-L	split one long line into two
ctrl-N	prepare disassembled line for easy editing
ctrl-O	Open space for character
ctrl-P	paste at cursor
ctrl-S	delete to screen end
ctrl-T	toggle insert mode for that line
ctrl-alt-D	delete until next space
ctrl-alt-K	delete to line start
shift left	home cursor, do NOT clear screen
shift down	insert line
shift right	delete line
shift up	home cursor, clear screen
return	execute line
enter	
ctrl-return	don't execute the line but go down to next one

The left mouse button can be used to reposition the cursor.

Special Keys

help	display commands,memory, (and 68030 registers)
esc	type non-ASCII characters
ctrl-B	blank the screen
ctrl-R	resume address placed on new line
alt-R	resume address pasted at cursor
ctrl-V	display ASCII table, color palettes, flag combos
ctrl-X	exchange KISSED and target screens, palettes, and resolutions
ctrl-W	display result of symbols, pc offset, breakpoints and $number in displayed disassembled opcode
ctrl-	"point and shoot" breakpoint
alt-3	toggle processor type and/or addressing mode format
alt-2	Displays new snapshot of the "window into memory"
ctrl-2	Displays old snapshot of the "window into memory"
ctrl-alt-2	Clears the snapshot of the "window into memory"

68000/68030	8, 9, 36, 37, 39, 46, 96, 100, 101
ASCII	8, 9, 14, 15, 20, 21, 23, 30, 60, 68, 69, 78, 99
(dis)assemble(r)	8, 9, 15, 20, 35-37, 49, 54, 58, 60-65, 67, 78, 95, 96, 99-101
block	13, 16, 17, 21, 22, 34, 35, 53, 54, 65, 68, 69, 71-75, 78, 92, 95
boolean	40, 51-53, 55, 99
breakpoint	7, 9, 13, 32, 34, 36, 38, 40-43, 45-49, 51-55, 57, 76, 78, 79, 93, 94, 96-99, 102
case	14, 56, 60
clone	92-94, 102
color	88, 93
cursor	6-11, 13, 20, 38, 42, 46, 49, 59, 62, 78, 93, 98-100
delimiter	21, 23, 30, 64, 69
debug(ger,ging)	46, 57, 64, 78, 95, 97
delete	6, 7, 11, 20, 62, 78, 88
digit	8, 14, 30, 38, 51, 52, 68, 84, 90, 93
edit(or)(ing)	6-8, 11-13, 20, 21, 23, 38, 42, 53, 60, 62, 68, 69, 71, 88, 91, 95
embed	39, 42, 85, 100
error	13, 14, 16, 34, 35, 54, 62, 64, 75, 80, 82, 84, 85, 93, 95
file	11, 35, 54, 60, 61, 75, 76, 78, 79, 90, 91, 94, 95, 101
fixed format	7, 39, 42, 64
floating point	83-86
hardware	25, 27, 28, 83
hilite	7-9, 11, 13, 14, 21, 25, 32, 36, 38, 40, 56, 62, 64, 71, 85, 90, 95
insert	6-8, 20, 97
instruction	9, 36, 40, 46, 48-51, 55, 62, 76, 94, 96-98, 100, 103
math	8, 15, 19, 39, 42, 68, 83, 85, 86, 100
memory	8, 9, 13, 15-21, 23, 25-35, 37, 44, 46, 53-55, 58,

	65, 68, 69, 71, 72, 74-79, 82, 83, 90-93, 95, 97-99
mouse	7, 14, 87
opcode	15, 36, 37, 43, 45, 58, 60, 62, 67, 96, 100
popup	8, 9, 32, 33, 46, 93, 97
print(er)	8, 14, 30, 87, 90, 91, 96
process	54, 76, 78, 80, 93, 94, 102, 103
processor	9, 13, 14, 36, 58, 60, 63
program	6, 9, 11, 32, 35, 36, 42, 44, 46, 47, 49, 51, 52, 54, 55, 58, 60, 61, 64, 67, 75-78, 80, 89, 92, 96-99
register	7, 8, 13, 33, 36-38, 42-46, 51-53, 55-57, 76, 96-100
return	10, 11, 13, 14, 20, 23, 32, 33, 38, 42, 45-48, 53, 56, 57, 69, 70, 78, 79, 87, 97, 98, 100
screen	6-8, 10, 11, 25, 32, 57, 58, 67, 90, 93, 96, 98
scroll	8, 10, 14, 21, 24, 57, 68, 69, 71, 87, 98
string	13-15, 23, 24, 30, 56, 64, 68, 69, 71, 98, 100
symbol	7, 9, 14, 30, 39, 40, 42, 51, 56, 58, 64, 66, 68, 76, 94, 99
target	8, 9, 32, 35, 36, 38, 42-44, 46-50, 54, 60, 61, 64, 75-78, 80, 89, 92, 95, 97-99, 103
trace	32, 36, 38, 43, 44, 46, 47, 48, 55, 56, 57, 80, 93, 97-99, 103
truncate	25, 26, 29, 53, 64
vector	80, 81, 99, 102, 103
width	21, 25-29, 32, 44, 71, 83, 84, 95
window	9, 32, 33, 97

Software: (Amiga only except as noted)

- MRBackup Professional - Harddrive backup software (packaged with our tape backup systems)
- Workbench Management Systems - Programmable buttons to execute your applications and scripts from workbench the intelligent way.
- RXTools - object oriented interface builder which extends the capabilities of ARexx(tm) on the Amiga, built-in editor
- Teachers Toolkit - a complete classroom management system, much more than just a guidebook or lesson planner
- Brigade Commander - real time, tactical war game with built-in unit/scenario editor, multi-screen maps, full digitized sound, animated weapon firing (IBM/Amiga)
- Brigade Data Disk ' The Red State' - Red State forces have taken over the U.S.A. Your job is to defeat the enemy forces and win back the United States.
- Memory Challenge Series I - education game for children ages 3 and up, strengthen recognition, memory retention and recall
- Thromulus- a game of microscopic conquest, use your Thromulus cells to surround and take over your opponents. one or two player. Called "Obsess-O-Matic" by .INFO magazine, this game will give hours of pure enjoyment!
- 4-Get-It! A new genre in strategy gaming! Match like tiles and clear each level. Includes multiple tiles including anit-grav, sticky and more! (IBM/Amiga)
- Under Pressure! - Arcade shoot'em up designed and programmed by the developers of Shadow of the Beast! 64 color screens, full motion animation and full stereo sound make this a must have!

Other TTR Development Products

TTR Development Inc. is proud to support the Amiga and MsDos computer line. We strive to provide a broad line of entertainment and productivity software and hardware for these impressive computer lines. We at TTR Development, Inc. appreciate your interest in our products and would like to keep you informed of other products currently available from TTR Development, Inc.

Hardware:

- Gemstone T150i/T150e internal/external Tandberg 150 Mb SCSI Streaming Tape System.

- Gemstone W150i/W150e internal/external Wangtek 150 Mb SCSI Streaming Tape System.

- Diamond Store 1300i/1300e internal/external Sony 1.3 Gb SCSI DAT System.

- Diamond Store 600i/600e Scd Laser Sony 600 Mb SCSI Magneto-optical System.

- Diamond Store 20 internal/external Insite Floptical Disk Drive for Amiga, Mac and IBM.

- Onyx Series Harddrive systems- Quantum low profile SCSI drives.

- Sapphire 020-68020/68881 Accelerators for A500, A1000 and A2000

- A100 Emerald Tower- attractive, small-footprint external device enclosure which can house up to 4 drives, ideal for our Onyx, Gemstone and Diamond Store lines.

Screen editing Merely display the current values and then cursor up and change the displayed value. One can even re-edit previous commands and use a paste feature.

Hilited changes Changes in registers and "windows in memory" during execution are hilited to make those changes immediately obvious.

Multitasking KISSED is well-behaved in a multitasking environment; KISSED and the target program are separate tasks (actually separate processes) which multitask. Thus KISSED can examine the target program "on the fly" if desired. KISSED can clone itself to simultaneously control up to 10 processes, including "child" processes spawned by the controlled "parent" process.

Trace buffer A variable sized trace buffer can be created in memory; besides the program counter, the other registers can also be saved and changes hilited.

Breakpoints Breakpoints have associated counters which can delay their activation until multiple passes through the location(s) have occurred. "Breakpoints" can even be set in the system ROMs. Boolean breakpoints can be set that only are activated if they pass an associated boolean test on some combination of the current register values, pc, memory, and/or constants. A test can also be made after each and every instruction for continuous testing.

Miniassembler/disassembler Symbols are fully supported by this pair as is screen editing. Thus, code can be disassembled and then reassembled merely by making changes in the displayed disassembly. Programs can also be reverse engineered by disassembly to disc, creating a source file that can be made ready for reassembly without major changes.

Execution options Besides the breakpoints mentioned above; tracing, execution until a "logic fork", a "point and shoot" breakpoint, a breakpoint after the current instruction, and a breakpoint "on the fly" are also all supported.

ETC. "Window into memory", unlimited nesting of embedded math, floating point math support, unlimited pointer indirection, etc